AFRL-RI-RS-TR-2015-125

INVESTIGATIONS OF FULLY HOMOMORPHIC ENCRYPTION (IFHE)

UNIVERSITY OF BRISTOL

MAY 2015

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2015-125 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /	**/ S /**
CARL R. THOMAS Work Unit Manager	MARK H. LINDERMAN Technical Advisor, Computing & Communications Division Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAY 2015	FINAL TECHNICAL REPORT	FEB 2011 – FEB 2015

4. TITLE AND SUBTITLE

INVESTIGATIONS OF FULLY HOMOMORPHIC ENCRYPTION (IFHE)

5a. CONTRACT NUMBER
FA8750-11-2-0079

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Nigel Smart

5d. PROJECT NUMBER
PROC

5e. TASK NUMBER
IF

5f. WORK UNIT NUMBER
HE

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Bristol
Department of Computer Science, Merchants Venturers Bldg., Woodland Rd
Bristol, United Kingdom, BS8-1UB

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

DARPA
675 N Randolph Street
Arlington VA 22203-2114

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2015-125

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The IFHE project (Investigation of Fully Homomorphic Encryption) originally set out to examine the security of FHE schemes, and the lattice hard problems on which they are based. This turned out to be relatively difficult, mainly due to the complexity of building software libraries which could support the advanced mathematics needed to perform experiments on modern multi-core processors. For example the NTL library is now (2015) able to support multi-threaded applications, but only if used with bleeding edge compilers on the latest Intel hardware. Thus the initial plan was perhaps a little ahead of its time.

However, by leveraging additional sources of funding; most notably from the UK's EPSRC and the EU's ERC, the Bristol team was able to make substantial headway in other areas related to the PROCEED program which were not originally envisaged. These are centred around;

• General techniques for Fully Homomorphic Encryption • Practical methods for actively secure Multi-Party Computation • General theory behind Multi-Party Computation. This report documents the progress made by the Bristol team.

15. SUBJECT TERMS

Fully Homomorphic Encryption, Secure Multi-Party Computation, Security, Privacy, Lattice

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**CARL R. THOMAS**
U	U	U	SAR	483	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Contents

Summary

The IFHE project (Investigation of Fully Homomorphic Encryption) originally set out to examine the security of FHE schemes, and the lattice hard problems on which they are based. This turned out to be relatively difficult, mainly due to the complexity of building software libraries which could support the advanced mathematics needed to perform experiments on modern multi-core processors. For example the NTL library is now (2015) able to support multi-threaded applications, but only if used with bleeding edge compilers on the latest Intel hardware. Thus the initial plan was perhaps a little ahead of its time.

However, by leveraging additional sources of funding; most notably from the UK's EPSRC and the EU's ERC, the Bristol team was able to make substantial headway in other areas related to the PROCEED programme which were not originally envisaged. These are centred around

- General techniques for Fully Homomorphic Encryption
- Practical methods for actively secure Multi-Party Computation
- General theory behind Multi-Party Computation

In this report we outline the various improvements and advances made by the team in Bristol.

Introduction

The PROCEED programme's goal was to investigate different methods for computing on encrypted data; in particular Fully Homomorphic Encryption and Multi-Party Computation. Over the course of the programme the IFHE team contributed a number of key advances in these two areas. The divide the contributions into four key sub-areas:

1. General techniques for Fully Homomorphic Encryption.
2. Practical methods for actively secure Multi-Party Computation.
3. General theory behind Multi-Party Computation.
4. Lattice based cryptanalysis.

Due to the ability to leverage additional funding, this report encompasses the whole of the activity in this space conducted by the Bristol team. For results for which DARPA funding was used to support the research we mark with three asterix's **** before the paragraph detailing the result. We feel that this will give the reader a better notion of how the research funded by DARPA fits within the overall portfolio of work in this space conducted in Bristol.

Perhaps the two key take home messages from the work conducted by the Bristol team are the greatly improved performance of actively secure MPC calculations; in

particular the development of the SPDZ protocol (described below), and the greatly improved practical performance of FHE schemes. These two advances are not unrelated, since the SPDZ protocol makes use of the advances in FHE schemes. Indeed one can see the SPDZ protocol as an example of where FHE technology can already be used to improve the performance of other security protocols.

Methods, Assumptions and Procedures

The work conducted is a mixture of traditional cryptographic theory work, and applied implementation work. This is a novel modus operandi, in that the Bristol group both works on the theoretical development of new protocols and schemes (along with their associated security proofs), and hand-in-hand works on building research prototypes to test the underlying performance of the resulting protocols. This is combined with a deep knowledge of pure mathematics (number theory in particular) which enables us to contribute to foundational work in the area.

This combination has allowed us to contribute a number of key ideas to the field over the course of the project:

- New techniques for Single Instruction Multiple Data (SIMD) operations of FHE schemes. These are based on the structure of rings of cyclotomic integers.
- New techniques for bootstrapping FHE schemes. We presented two different techniques for this; one based on extending earlier work on FHE schemes to plaintext spaces embedded p-adic rings, and one to the use of different group representations.
- Parameter size analysis for key generation. This has been key to developing the instantiation of techniques for the SPDZ protocol suite.
- Our protocol design work has focused on efficient covertly secure offline processing for the SPDZ protocol and to algorithms to implement fast online functionalities; for example floating point calculations and ORAM access.

Results and Discussion

We discuss each of the four areas mentioned above in turn:

General techniques for Fully Homomorphic Encryption.

Much of our initial work in PROCEED centred around the development of FHE techniques. In this work we focused on developing new ways of utilizing FHE techniques to enable faster and more elaborate computations. In other work, described in later sections, we applied these FHE techniques to enable faster MPC protocols, and we examined the security of the underlying lattice based cryptosystems.

*** The first output from our DARPA funded work on FHE was the development of a method for SIMD evaluation for the original Gentry FHE scheme, [1]. Being journal

published the paper took many years to appear in final form. The paper showed how Gentry's original scheme [2], as optimized by Smart and Vercauteren [3], could be modified to support the operation on many plaintext elements at once. In addition to this key finding, the authors also proposed a method to perform bootstrapping in SIMD parallelism. The work in this paper has been highly influential and the basic idea has been exploited in all implementations of FHE schemes since. Although much of the specific techniques are now less important since there are better schemes than the original Gentry scheme now.

*** In order to support the above SIMD operations new key generation techniques were needed for the FHE scheme; these were introduced in [4, 5], which built on earlier work in [6]. In particular the usage of Fast Fourier Transform techniques were used to simplify the key generation step for parameters in the Smart-Vercauteren variant of Gentry's FHE scheme, in order to enable SIMD operation of the scheme.

*** In [5] we presented attacks on the SHE schemes at the time in a model in which the attacker had access to a decryption oracle, before any challenge ciphertext was provided. Since all known FHE schemes include a decryption hint within the public key, this means that we were restricted to SHE schemes. In addition since SHE schemes are malleable only so-called lunch-time chosen ciphertext attacks were analysed. We presented a number of attacks, and showed how one particular scheme could be immunised against such attacks using a novel lattice based knowledge assumption.

*** Our focus on FHE then turned to a series of papers with Gentry and Halevi on the BGV FHE scheme [7]. This scheme, based on Ring-LWE, supports the SIMD vector operations described above; but it is both more efficient and based on a harder problem than the initial FHE schemes discussed above. In our first work on this scheme in [8], we described how combining the SIMD addition and multiplication operations, with permutation operations induced from the Galois group of the underlying number field, enabled us to produce asymptotically efficient FHE schemes. Whilst mainly theoretical in nature, the introduction of the concept of homomorphically applying Galois action to the encrypted plaintext has turned out to be highly important in practice for obtaining efficient general homomorphic operations.

*** For plaintext rings in characteristic two, the Galois group not only provides a mechanism to apply permutations to the plaintext slots, it also provides the Frobenius automorphism; which enables very fast powering by powers of two. This was exploited in our next paper [9], which presented the first large scale computation performed using SHE/FHE technology. We showed that the evaluation of a circuit as complex as that of the AES function was possible; albeit rather slowly. The use of AES as a test bench circuit for computations on encrypted data was introduced by myself in 2009 in [10]. In subsequent works various authors have been able to homomorphically evaluate the AES circuit in under five minutes; which is remarkable

given the state of the art at the start of the PROCEED programme. This paper won the IBM Pat Goldberg Award for Best Paper in Computer Science for 2012.

*** In our next paper [11] Gentry, Halevi, and myself, turned our attention to how to perform efficient bootstrapping of BGV ciphertexts. We utilized a special ciphertext modulus, which close to a power of the plaintext modulus, so as to provide an algebraic decryption operation (as opposed to the circuit based approaches of earlier works). This enabled a more efficient procedure. In extending the technique to bootstrapping SIMD encryptions we required the development of techniques to efficiently homomorphically evaluate Fourier Transforms.

*** Motivated by the need to perform homomorphic Fourier transforms, we then turned our attention to a technique to homomorphically switch from one ring to another. However, it turned out that such a technique would have wider applicability in that it enabled more efficient noise management via choosing different rings at different points in the computation. Thus, with Gentry, Halevi and Peikert, we developed a complete theory of this operation which was described in [12] and [13].

*** In very recent work [14] myself and two members of my group develop a new novel bootstrapping technique for BGV ciphertexts which has lower depth than all previous techniques. The methodology makes use of the general representation technique of [11], but it uses a new way of representing the various groups under consideration. It is unclear at present whether this new technique will be practically relevant, since the decrease in depth is paid for by an increase in the computational complexity (i.e. the number of multiplications).

Outside of the DARPA project, a student in my group, working with colleagues from Microsoft Research in Redmond, developed an improved variant of the NTRU based FHE scheme [15]. The paper presents a number of optimizations of the NTRU based scheme, as well as implementation results.

Practical methods for actively secure Multi-Party Computation.

Probably the most important results from the IFHE project was the development of the SPDZ protocol; this is an n-party MPC protocol which is actively/covertly secure. It is in the pre-processing model, and the pre-processing utilizes the SIMD optimizations of the BGV FHE scheme as described above in [8]. After the development of the basic protocol, our work (funded mainly by the EPSRC and ERC) focused on building a large MPC system based on the basic protocol. In the following paragraphs we elaborate on the various optimizations and improvements obtained.

*** This entire line of work started with the joint work with Aarhus University explained in [16]. This paper took a number of ideas from earlier MPC protocols developed by Aarhus (namely the use of pre-processing and MACs to obtain active

security), and greatly improved the overall efficiency and practicality of the methods. As mentioned above a key innovation was the use of FHE technology as a means to obtain a performance improvement over protocols which did not utilize FHE technology; thus this is probably the first example of where FHE technology developed in PROCEED was used to improve performance of a security protocol.

In SPDZ, online circuit evaluations are done via secret-sharing the inputs, and having each party evaluating the circuit almost locally in his shares. This can be done very efficiently, with only multiplication requiring the communication of two field elements. All circuit value are augmented with a message authentication code (MACs). Parties communicating, or using incorrect values in his local evaluation, will be detected by the other parties. Previous MAC schemes required each party linear storage in the total number of participants. SPDZ brings it down to a constant.

*** As an early test of the SPDZ system we implemented a system to evaluate the AES functionality [17]; again the choice of AES as a test case was due to our proposing this in [10]. The initial results were relatively good, and comparable to systems with a weaker security guarantee. However, now the run times can be considerably improved.

*** The preprocessing of SPDZ relies on somewhat homomorphic encryption (SHE). This SHE scheme allows one to homomorphically add a number of ciphertexts, and to perform a single homomorphic multiplication. This is in constrast with fully homomorphic encryption, which allows an unbounded number of multiplications. Key open problems in the initial paper [16] was that the protocol did not enable reactive computation, that no procedure was given to agree the FHE public/private key pair. These problems were solved in [18] where a method was given for the parties to agree in a cryptographic key. Also, support for reactive computations was given; exploiting the secret-sharing approach, it was shown how to check MACs without having to reveal the key for this check; hence, after one single online computation is done, the participants can carry on in a different computation, with the remaining authenticated entropy generated in the preprocessing with the secret MAC key. The online evaluations can even occur concurrently, since SPDZ operates in the UC security framework. This paper won the Best Paper Award at ESORICS 2013.

The main advantage of the SPDZ protocol is its very efficient online phase, which only requires standard symmetric and information theoretic primitives to implement. Since this is the only part of the protocol dependent on the parties' inputs to the function, the running time of the online phase determines the latency a user experiences when waiting for the output, and hence is crucial to implement in the most efficient way possible. Moreover, to be able to implement complex functions in MPC we need a suitable set of tools to compile and run programs written in some high-level language. To do this, we designed and implemented an MPC virtual machine for the online phase, which parses and executes a special MPC-based set of basic instructions. We then created a compiler that reads Python-like programs

and performs various optimizations to output efficient bytecode that can be run by the VM. One of the key optimizations is to minimize the number of rounds of communication in a given program by analyzing the control flow graph, which greatly reduces latency. Using this toolchain, we created very efficient implementations of common functions including crypto-specific benchmarks AES and SHA-1, as well as other functions such as sorting and floating point arithmetic, which have applications to more general scenarios. The entire system is described in [19]. We are continuing to improve and extend the features of the compiler, and in the future would like a system that can formally verify the correctness and security of protocols.

Traditionally, MPC only allows one to execute binary or arithmetic circuits. This makes advanced data structures, such as arrays, very inefficient because one has to scan the entire array for every access. A technique called oblivious RAM (ORAM) facilitates more efficient data structures in MPC. The scope of ORAM goes beyond MPC, generally hiding the access pattern in a client-server model. A recent result on ORAM much simplified the necessary computation. This allowed for the first implementation of arrays and priority queues in secret-sharing MPC [20]. Both are used the Bristol implementation of an algorithm for shortest paths in a graph (Dijkstra's algorithm), which is significantly faster than previous implementations. For our implemention, we had to improve various aspects of our platform, for example the support of non-recursive functions and better parallelization. Future research in this area will focus on implementing general RAM programs in MPC and aspects thereof such as cost-privacy trade-offs.

*** Whilst SPDZ is highly suited to arithmetic circuits, it is less well suited to binary circuits. For binary circuits the best protocol seems to be TinyOT [21]. However, TinyOT is only suited to two players. In [22] we extended the TinyOT protocol to the multi-party case. The protocol we describe allows active secure evaluation of Boolean circuits in the dishonest majority setting with static corruptions. The idea is that of using an information-theoretic MAC applied to the oblivious transfer (OT) based GMW protocol, and producing in the offline phase a large number of random authenticated OTs, which are then used to perform Beaver's style multiplications in the online phase. The efficiency of the offline phase is guaranteed by a variant of an OT-extension protocol. The main tool we use is an extension of the authenticated Bit (aBit) primitive from [21] from the two-party to the multi-party setting, that is obtained combining, in a nontrivial way, ideas from [21], [18] and [16]. In particular, we use a global unknown shared key instead of pairwise keys for bits authentication, and then, by executing the pairwise aBit protocol, we are able to obtain secret shared random bits, together with shared MACs, by all n-parties.

After publication we realised that the paper [22] contained a minor bug, we are currently working with the Aarhus group on a joint paper which merges the work in [21] and [22], and corrects the bug in the published version of [22].

General theory behind Multi-Party Computation.

As well as the more practical aspects of MPC, we also examined more theoretical aspects. Much of the work done in this area was by my two post-docs Choudhury and Patra who were funded by EPSRC; and have since returned to Bangalore where they now have permanent academic positions.

*** In [23] we examined the situation of a server farm with thousands of nodes, which wanted to run an MPC calculation where a given (small'ish) percentage are corrupt. We present a protocol which does not require full communication between all nodes at all times, yet still obtains full active, and robust, security. This is done via a sequence of checkpoints, and then running between the checkpoints, an actively secure dishonest majority protocol between a suitably large sub-committee. By selecting the dishonest majority sub-protocol so that we can detect cheaters we are then able to apply standard player elimination strategies so as to obtain an overall robust protocol.

*** On one hand FHE allows us to perform computation on encrypted data using very little communication but a lot of computational resources; whereas standard MPC protocols require little computational resources, but a lot of computation. In [24] we presented a technique which interpolates between the FHE-MPC protocol of Gentry and more standard MPC protocols. The protocol enables one to select a depth of sub-circuit which is dealt with via the FHE part, and the rest is done via an MPC protocol. This division of the circuit into layers is reminiscent of the previous paper [23].

*** In most MPC protocols one assumes that the function to be computed is public, and hence known to all parties. However, there are some situations where one might want to keep the function private. Treating the function as one players input is clearly captured by an MPC protocol which enables one to implement a Universal Turing machine. Thus this problem is purely one of efficiency. In [25] a protocol is given, which is essentially optimal, in the case of active adversaries. Active security is obtained via the use of MACs, like the SPDZ protocol, however the underlying MPC protocol is very different in nature.

Related to the PROCEED programme was a series of papers by my post-docs Choudhury and Patra on aspects of MPC in the case of asynchronous networks. Almost all practical MPC protocols assume that the underlying network is synchronous, however real networks are asynchronous. This is a particular problem in MPC as a receiver will never know if the fact he did not receive a message is due to network issues, or a corrupted sender. Thus there is a whole sub-area of MPC research (currently mostly theoretical) which deals with issues related to asynchronicity. Bristol's work in this area during the time of the PROCEED programme resulted in the following outputs [26] [27] [28] [29] [30] [31]

Finally in [32] secure two-party computation with single adaptive corruptions in the nonerasure model where at most one party is adaptively corrupted was studied. To distinguish this notion from fully adaptive security, where both parties may get corrupted, we denote it by one-sided adaptive security. Our goal in this work is to make progress in the study of the efficiency of two-party protocols with one sided security. Our measure of efficiency is the number of public key encryption operations. Loosely speaking, our primitives are parameterized by a public key encryption scheme for which we count the number of key generation, encryption and decryption operations. More concretely, these operations are captured by the number of exponentiations in several important groups.

Lattice based cryptanalysis.

Lattice-based cryptography is one of the main candidates for cryptography that remains secure against cryptanalysis using a quantum computer. For some time now, cryptographers and quantum algorithms researchers alike have not found any quantum attack that provides a speed-up similar to Shor's algorithm for RSA and Discrete Log. However, Grover's algorithm is also important for cryptography, as it implies that we need to use keys that are twice as long in the symmetric setting for example. Of particular relevance to PROCEED is the fact that all FHE schemes currently known rely for their security on the hardness of various lattice based problems.

In [33], my students examined the effects of using Grover inside the so-called sieving algorithms for solving the shortest vector problem in lattices. Previously, there had been one work using Grover on a single (and different) algorithm, but we extend this to the whole class of sieving algorithms. Our analysis shows that the application of Grover allows sieving-type algorithms to be asymptotically faster than the best classical algorithms (which do not use sieving). As a rule of thumb, it appears that the keys need to be increased by a factor 4/3 to achieve the same classical level of security.

*** My student and I further examined the relation between different parameters and security in [34], but for the best known classical of algorithm. Previous work had always discarded the dimension parameter as a second order term for security. However, we are interested in FHE schemes, where the dimension needs to be huge for the functionality of the scheme. We designed an approach that takes the dimension into account as well, which led to a decrease in parameters for the same security when we applied it to several FHE schemes.

Lattices are not only important for the construction of cryptography. The history of lattices in cryptanalysis stretches even further, all the way back to the breaking of knapsack-based schemes and breaking RSA with partial information on the key. It

turns out that lattices are quite good at extending lots of different instances of partial information into the full information, e.g., the secret key.

*** In joint work with colleagues from Adelaide [35] we combined a side-channel attack on ECDSA in OpenSSL with lattice algorithms to recover the secret key. We modified the previous approaches to combine instances where the amount of partial information varies per instance. Our results show that such attacks can be very efficient, both in terms of the number of signatures required and the time required.

Conclusion

As can be seen the IFHE project has created a large number of outputs in a range of topics related to the PROCEED programme. The main outputs have shown how computation on encrypted data is now much closer to a deployable protocol compared to the state of the art at the start of the programme. We have identified areas in which FHE can be used as a performance enhancing technology (e.g. the SPDZ protocol), and we have shown that active security can be achieved for MPC protocols with very little performance overhead compared to passively secure protocols.

Our work in this space has attracted considerable interest from partners outside of the PROCEED programme. Follow up work is continuing in the EU funded PRACTICE project on MPC, and in the HEAT project on FHE. We have also conducted a joint project with Thales funded by the UKs DSTL into applications of MPC with in the UK defence sector. Finally, with Bar-Ilan University we have formed a company Dyadic Security which is looking at applications of MPC to breach mitigation on computer networks.

References

[1] N. P. Smart and F. Vercauteren, “Fully Homomorphic SIMD Operations,” *Designs, Codes and Cryptography,* vol. 71, pp. 57-81, 2014.

[2] C. Gentry, A fully homomorphic encryption scheme., PhD Thesis, Stanford University, 2009.

[3] N. P. Smart and F. Vercauteren, “Fully homomorphic encryption with relatively small key and ciphertext sizes,” in *Public Key Cryptography - PKC 2010*, 2010.

[4] P. Scholl and N. P. Smart, “Improved Key Generation for Gentry's Fully Homomorphic Encryption Scheme,” in *Cryptography and Coding - IMACC 2011*, 2011.

[5] J. Loftus, A. May, N. P. Smart and F. Vercauteren, “On CCA-Secure Somewhat Homomorphic Encryption,” in *Selected Areas in Cryptography - SAC 2011*, 2012.

[6] C. Gentry and S. Halevi, “Implementing Gentry's fully-homomorphic encryption scheme,” in *Advances in Cryptology - EUROCRYPT 2011*, 2011.

[7] Z. Brakerski, C. Gentry and V. Vaikuntanathan, “(Leveled) fully homomorphic encryption without bootstrapping.,” in *ACM, Innovations in Theoretical Computer Science 2012*, 2012.

[8] C. Gentry, S. Halevi and N. P. Smart, “Fully homomorphic encryption with polylog overhead,” in *Advances in Cryptology - EUROCRYPT 2012*, 2012.

[9] C. Gentry, S. Halevi and N. P. Smart, “Homomorphic evaluation of the AES circuit,” in *Advances in Cryptology - CRYPTO 2012*, 2012.

[10] B. Pinkas, T. Schneider, N. P. Smart and S. C. Williams, “Secure two-party computation is practical,” in *Advances in Cryptology - AsiaCrypt 2009*, 2009.

[11] C. Gentry, S. Halevi and N. P. Smart, “Better Boostrapping in Fully Homomorphic Encryption,” in *Public Key Cryptography - PKC 2012*, 2012.

[12] C. Gentry, S. Halevi, C. Peikert and N. P. Smart, “Ring Switching in BGV-Style Homomorphic Encrypton,” in *Security and Cryptography for Networks - SCN 2012*, 2012.

[13] C. Gentry, S. Halevi, C. Peikert and N. P. Smart, “Field Switching in BGV-Style Homormorphic Encryption,” *Journal of Computer Security,* pp. 663-684, 2013.

[14] E. Orsini, J. van de Pol and N. P. Smart, “Bootstrapping BGV Ciphertexts with a Wider Choice of p and q,” in *To appear Public Key Cryptography - PKC 2015*, 2015.

[15] J. Bos, K. Lauter, J. Loftus and M. Naehrig, “Improved security for a ring-based Fully Homomorphic Encryption scheme,” in *Cryptography and Coding - IMACC 2013*, 2013.

[16] I. Damgård, V. Pastro, N. P. Smart and S. Zakarias, “Multiparty computation from somewhat homomorphic encryption,” in *Advances in Cryptology - CRYPTO 2012*, 2012.

[17] I. Damgård, M. Keller, E. Larraia, C. Miles and N. P. Smart, “Implementing AES via an actively/covertly secure dishonest majority MPC protocol,” in *Security and Cryptography for Networks - SCN 2012*, 2012.

[18] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl and N. P. Smart, “Practical covertly secure MPC for dishonest majority,” in *ESORICS 2014*, 2013.

[19] M. Keller, P. Scholl and N. P. Smart, “An architecture for practical actively secure MPC with dishonest majority,” in *ACM-CCS 2013*, 2013.

[20] M. Keller and P. Scholl, “Efficient Oblivious Data Structures for MPC,” in *Advances in Cryptology - AsiaCrypt 2014*, 2014.

[21] J. Nielsen, P. Nordholt, C. Orlandi and S. Burra, “A new approach to practical active-secure two-party computation,” *Crypto 2012,* vol. Springer LNCS 7417, pp. 681-700, 2012.

[22] E. Larraia, E. Orsini and N. P. Smart, “Dishonest majority Multi-Party Computation for Binary Circuits,” in *Advances in Cryptology - CRYPTO 2014*, 2014.

[23] A. Choudhury, A. Patra and N. P. Smart, “Reducing the overhead of MPC over a large population,” in *Security and Cryptography for Networks - SCN 2014*, 2014.

[24] A. Choudhury, J. Loftus, E. Orsini, A. Patra and N. P. Smart, “Between a Rock and a Hard Place: Interpolating between MPC and FHE,” in *Advances in Cryptology - AsiaCrypt 2013*, 2013.

[25] P. Mohassel, S. Sadeghian and N. P. Smart, “Actively secure private function evaluation,” in *Advances in Cryptology - AsiaCrypt 2014*, 2014.

[26] A. Choudhury and A. Patra, “On the communication complexity of reliable and secure message transmission in asynchronous networks.,” in *Information Security and Cryptology - ICISC 2011*, 2011.

[27] A. Choudhury and A. Patra, “Brief Announcement: Efficient Optmally Resilient Statistical AVSS and its Applications,” in *ACM PODC 2012*, 2012.

[28] A. Choudhury, “Breif Announcement: Optimal Amortized Secret Sharing with Cheater Identification,” in *ACM PODC 2012*, 2012.

[29] A. Badanidiyuru, A. Patra, A. Choudhury, S. Kannan and R. Pandu, “On the trade-off between network connectivity, round complexity and communication complexity of reliable message transmission,” *Journal of the ACM,* vol. 22, pp. 1-35, 2012.

[30] A. Choudhury, “Breaking the O(n |C|) barrier for unconditionally secure asynchronous multiparty computation,” in *Topics in Cryptology - INDOCRYPT 2013*, 2013.

[31] A. Choudhury, M. Hirt and A. Patra, “Asynchronous multiparty computation with linear communication complexity,” in *DISC 2013*, 2013.

[32] C. Hazay and A. Patra, “One-sided adaptively secure two-party computation,” in *Theory of Cryptography - TCC 2014*, 2014.

[33] T. Laarhoven, M. Mosca and J. van de Pol, “Solving the shortest vector problem in lattices faster using quantum search.,” in *Post-Quantum Cryptography - PKC 2013*, 2013.

[34] J. van de Pol and N. P. Smart, “Estimating key sizes for high dimensional lattice-based systems,” in *Coding and Cryptography - IMACC 2013*, 2013.

[35] N. Benger, J. van de Pol, N. P. Smart and Y. Yarom, “"Ooh Ahh... Just a Little Bit": A small amount of side channel can go a long way,” in *Cryptographic Hardware and Embedded Systems - CHES 2014*, 2014.

List of Symbols, Abbreviations and Acronyms

AES	Advanced Encryption Standard
BGV	Brakerski-Gentry-Vaikuntanathan FHE Scheme
FHE	Fully Homomorphic Encryption
LWE	Learning With Errors
MPC	Multi-Party Computation
ORAM	Oblivious Random Access Memory
SHE	Somewhat Homomorphic Encryption
SIMD	Single Instruction Multiple Data
SPDZ	Smart-Pastro-Damgard-Zakarais MPC Protocol

Appendix

The following appendix contains the papers which are the output of the project. For each paper we give the *full version* from the IACR ePrint Archive and not the extended abstract (which is the one usually published in conferences).

Noname manuscript No.
(will be inserted by the editor)

Fully Homomorphic SIMD Operations

N.P. Smart · F. Vercauteren

the date of receipt and acceptance should be inserted later

Abstract At PKC 2010 Smart and Vercauteren presented a variant of Gentry's fully homomorphic public key encryption scheme and mentioned that the scheme could support SIMD style operations. The slow key generation process of the Smart–Vercauteren system was then addressed in a paper by Gentry and Halevi, but their key generation method appears to exclude the SIMD style operation alluded to by Smart and Vercauteren. In this paper, we show how to select parameters to enable such SIMD operations. As such, we obtain a somewhat homomorphic scheme supporting both SIMD operations and operations on large finite fields of characteristic two. This somewhat homomorphic scheme can be made fully homomorphic in a naive way by recrypting all data elements seperately. However, we show that the SIMD operations can be used to perform the recrypt procedure in parallel, resulting in a substantial speed-up. Finally, we demonstrate how such SIMD operations can be used to perform various tasks by studying two use cases: implementing AES homomorphically and encrypted database lookup.

1 Introduction

For many years a long standing open problem in cryptography has been the construction of a fully homomorphic encryption (FHE) scheme. The practical realisation of such a scheme would have a number of consequences, such as computation on encrypted data held on an untrusted server. In 2009 Gentry [10,11] came up with the first construction of such a scheme based on ideal lattices. Soon after Gentry's initial paper appeared, two other variants were presented [6,23]; the method of van Dijk et al. [6] is a true variant of Gentry's scheme and relies purely on the arithmetic of the integers; on the other hand the scheme of Smart and Vercauteren [23] is a specialisation of Gentry's scheme to a particular set of parameters.

All schemes make use of Gentry's idea of first producing a somewhat homomorphic encryption scheme and then applying a bootstrapping process to obtain a complete FHE scheme. This bootstrapping process requires a "dirty" ciphertext to be publicly reencrypted into a "cleaner" ciphertext. This requires that the somewhat homomorphic scheme can homomorphically implement its own decryption circuit, and so must be able to execute a circuit of a given depth.

Gentry and Halevi [12] presented an optimized version of the Smart–Vercauteren variant. In particular, the optimized version has an efficient key generation procedure based on the Fast Fourier Transform and a simpler decryption circuit. These two major optimizations, along with some other minor ones, allow Gentry and Halevi to actually implement a "toy" FHE scheme, including the ciphertext cleaning operation.

N.P. Smart
Dept. Computer Science,
University of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB,
United Kingdom.
E-mail: nigel@cs.bris.ac.uk

F. Vercauteren
COSIC - Electrical Engineering,
Katholieke Universiteit Leuven,
Kasteelpark Arenberg 10,
B-3001 Heverlee,
Belgium.
E-mail: fvercaut@esat.kuleuven.ac.be

Smart and Vercauteren mentioned in [23] that their scheme can be adapted to support SIMD (Single-Instruction Multiple-Data) style operations on non-trivial finite fields of characteristic two, as opposed to operations on single bits, as long as the parameters are chosen appropriately. However, the parameters proposed in both [12] and [23] do not allow such SIMD operations, nor direct operation on elements of finite fields of characteristic two of degree greater than one. In particular, the efficient key generation method of [12] precludes the use of parameters which would support SIMD style operations. Using fully homomorphic SIMD operations would be an advantage in any practical system since FHE schemes usually embed relatively small plaintexts within large ciphertexts. Allowing each ciphertext to represent a number of independent plaintexts would therefore enable more efficient use of both space and computational resources.

In this paper we investigate the use of SIMD operations in FHE systems in more depth. In particular we show how by adapting the parameter settings of [12,23] one can obtain the benefits of SIMD operations, whilst still maintaining many of the important efficiency improvements obtained by Gentry and Halevi. We thus obtain a somewhat homomorphic scheme supporting SIMD operations, and operations on large finite fields of characteristic two. We then discuss how one can use the SIMD operations to perform the recrypt procedure in parallel. In addition we explain how such SIMD operations could be utilized to perform a number of interesting higher level operations, such as performing AES encryption homomorphically and searching an encrypted database on a remote server.

The paper is structured as follows. Section 2 presents some basic facts about finite fields and algebras defined as quotients of polynomial rings. Section 3 explains how these algebras allow us to create a somewhat homomorphic encryption scheme whose message space consists of multiple parallel copies of a given finite field of characteristic two. Section 4 describes a recryption procedure for the somewhat homomorphic scheme that preserves the underlying message space structure. Section 5 contains our main contribution, namely, a recryption procedure that makes use of the SIMD operations. This new procedure significantly reduces the cost of recryption. To justify our claims, Section 7 presents implementation timings for a toy example. Finally, Section 8 gives possible applications of the SIMD structure of our FHE scheme, including bit-sliced implementations of algorithms, such as performing AES encryption using an encrypted key, and database search.

Since the appearance of the current paper on IACR e-Print in March 2011 the basic idea of utilizing SIMD operations has been used by a number of authors, and the methods in this paper have been extended. In particular in [22] the authors present further optimizations of the key generation method proposed in this paper. It had already been noted in [2] that the ring-LWE based FHE schemes also posses exactly the same form of SIMD operation in this paper. In a series of work [13–16] Gentry, Halevi and Smart make extensive use of FHE based SIMD operations in a number of contexts related to the BGV cryptosystem [2]. In [13] they show how using SIMD operations combined with the BGV scheme allows one to obtain an asymptotically efficient FHE scheme; then in [14] they show (among other results) how a SIMD evaluation of the FFT transform can be used to possibly improve bootstrapping functionality; then in [15] they actually implement the example application we present in Section 8.2; finally in [16] they show how one can in SIMD switch the underlying finite field over which one is working to a smaller one, thus obtaining performance improvements as one descends via a levelled FHE scheme. In [9] the authors also utilize the SIMD mode of the basic somewhat homomorphic BGV scheme to achieve a higher efficient offline phase for a multi-party computation protocol.

1.0.1 Notations

We end this introduction by presenting the notations that will be used throughout this paper. Assignment to variables will be denoted by $x \leftarrow y$. If A is a set then $x \leftarrow A$ implies that x is selected from A using the uniform distribution. If A is an algorithm then $x \leftarrow A$ implies that x is obtained from running A, with the resulting probability distribution being induced by the random coins of A. For integers x, d, we denote $[x]_d$ the reduction of x modulo d into the interval $[-d/2, d/2)$. If $\mathbf{y}$ is a vector then we let $\mathbf{y}_i$ denote the i'th element of $\mathbf{y}$.

Polynomials over an indeterminate X will (usually) be denoted by uppercase roman letters, e.g. $F(X)$. We make an exception for the cyclotomic polynomials which are as usual denoted by $\Phi_m(X)$. Elements of finite fields and number fields defined by a polynomial $F(X)$, i.e. elements of $\mathbb{F}_2[X]/F(X)$ and $\mathbb{Q}[X]/F(X)$, can also be represented as polynomials in some fixed root of $F(X)$ in the algebraic closure of the base field. We shall denote such polynomials by lower case greek letters, with the fixed root (being an element of the field) also being denoted by a lower case greek letter; for instance $\gamma(\theta)$ where $F(\theta) = 0$. When the underlying root of $F(X)$ is clear we shall simply write γ.

For a polynomial $F(X) \in \mathbb{Q}[X]$ we let $\|F(X)\|_\infty$ denote the ∞-norm of the coefficient vector, i.e. the maximum coefficient in absolute value. Similarly, for an element $\gamma \in \mathbb{Q}[X]/F(X)$ we write $\|\gamma\|_\infty$ for $\|\gamma(X)\|_\infty$ where $\gamma(X)$ is the corresponding unique polynomial of degree $< \deg(F)$. If $F(X) \in \mathbb{Q}[X]$ then we let $\lceil F(X) \rfloor$ denote the polynomial in $\mathbb{Z}[X]$ obtained by rounding the coefficients of $F(X)$ to the nearest integer. Similary, for an element $\gamma \in \mathbb{Q}[X]/F(X)$ we write $\lceil \gamma \rfloor$ for $\lceil \gamma(X) \rfloor$.

2 Fields and Homomorphisms

To present the SIMD operations in full generality and to understand how they can be utilized we first set up a number of finite fields and homomorphisms between them. We let $F(X) \in \mathbb{F}_2[X]$ denote a monic polynomial of degree N that we assume to split into exactly r *distinct* irreducible factors of degree $d = N/r$

$$F(X) := \prod_{i=1}^{r} F_i(X).$$

In practice $F(X)$ will be the reduction modulo two of a specially chosen monic *irreducible* polynomial over $\mathbb{Z}$. This polynomial $F(X)$ defines a number field $\mathbb{K} = \mathbb{Q}(\theta) = \mathbb{Q}[X]/(F)$, where θ is some fixed root in the algebraic closure of $\mathbb{Q}$.

Let A denote the algebra $A := \mathbb{F}_2[X]/(F)$, then by the Chinese Remainder Theorem we have the natural isomorphisms

$$\begin{aligned} A &\cong \mathbb{F}_2[X]/(F_1) \otimes \cdots \otimes \mathbb{F}_2[X]/(F_r), \\ &\cong \mathbb{F}_{2^d} \otimes \cdots \otimes \mathbb{F}_{2^d}, \end{aligned}$$

i.e. A is isomorphic to r copies of the finite field $\mathbb{F}_{2^d}$. Arithmetic in A will be defined by polynomial arithmetic in the indeterminate X modulo the polynomial $F(X)$. Our goal in this section is to relate arithmetic in A explicitly with the elements in subfields of the $\mathbb{F}_{2^d}$.

We let θ_i denote a fixed root of $F_i(X)$ in the algebraic closure of $\mathbb{F}_2$. To aid notation we define $\mathbb{L}_i := \mathbb{F}_2[X]/(F_i)$ and note that all the $\mathbb{L}_i$ are isomorphic as fields, where the isomorphisms are explicitly given by

$$\Lambda_{i,j} : \begin{cases} \mathbb{L}_i & \longrightarrow & \mathbb{L}_j \\ \alpha(\theta_i) & \longmapsto & \alpha(\rho_{i,j}(\theta_j)), \end{cases}$$

with $\rho_{i,j}(\theta_j)$ a fixed root of F_i in $\mathbb{L}_j$, i.e. we have $F_i(\rho_{i,j}(X)) \equiv 0 \pmod{F_j(X)}$.

For each divisor n of d, the finite field $\mathbb{K}_n := \mathbb{F}_{2^n}$ is contained in $\mathbb{F}_{2^d}$. We assume a fixed canonical representation for $\mathbb{K}_n$ as $\mathbb{F}_2[X]/K_n(X)$ for some irreducible polynomial $K_n(X) \in \mathbb{F}_2[X]$ of degree n, which is often fixed by the application. We let ψ denote a fixed root of $K_n(X)$ in the algebraic closure of $\mathbb{F}_2$. Since $\mathbb{K}_n$ is contained in each of $\mathbb{L}_i$ defined above, we have explicit homomorphic embeddings given by

$$\Psi_{n,i} : \begin{cases} \mathbb{K}_n & \longrightarrow & \mathbb{L}_i \\ \alpha(\psi) & \longmapsto & \alpha(\sigma_{n,i}(\theta_i)), \end{cases}$$

with $\sigma_{n,i}(\theta_i)$ a fixed root of $K_n(X)$ in $\mathbb{L}_i$, i.e. $K_n(\sigma_{n,i}(X)) \equiv 0 \pmod{F_i(X)}$. Note that the above mapping is linear in the coefficients of $\alpha(\psi)$.

Combining the above homomorphic embedding with the Chinese Remainder Theorem, we obtain a homomorphic embedding of $l \leq r$ copies of $\mathbb{K}_n$ into the algebra A via

$$\Gamma_{n,l} : \begin{cases} \mathbb{K}_n^l & \longrightarrow & A \\ (\kappa_1(\psi), \ldots, \kappa_l(\psi)) & \longmapsto & \sum_{i=1}^{l} \kappa_i(\sigma_{n,i}(X)) \cdot H_i(X) \cdot G_i(X), \end{cases}.$$

The polynomials $H_i(X)$ and $G_i(X)$ are given by the Chinese Remainder Theorem and are defined as

$$H_i(X) \leftarrow F(X)/F_i(X) \text{ and } G_i(X) \leftarrow 1/H_i(X) \pmod{F_i(X)}.$$

We shall denote component wise addition and multiplication of elements in $\mathbb{K}_n^l$ by $\mathbf{k}_1 + \mathbf{k}_2$ and $\mathbf{k}_1 \times \mathbf{k}_2$. As such we have constructed two equivalent methods of computing with elements in $\mathbb{K}_n^l$: the first method simply computes component wise on vectors of l elements in $\mathbb{K}_n$, whereas the second method first maps all inputs to the algebra A using $\Gamma_{n,l}$, performs computations in A and finally maps back to $\mathbb{K}_n^l$ via $\Gamma_{n,l}^{-1}$. Note that by construction $\mathbb{K}_n^l$ and $\Gamma_{n,l}(\mathbb{K}_n^l)$ are isomorphic, so that $\Gamma_{n,l}^{-1}$ is always well defined on the result of the computation.

The goal of this paper is to produce a fully homomorphic encryption scheme that allows us to work via SIMD operations on l copies of $\mathbb{K}_n$ at a time, for all n dividing d, by computing in the algebra A. In particular, this enables us to support SIMD operations both in $\mathbb{F}_2$ and $\mathbb{F}_{2^d}$. To make things concrete the reader should consider the example of $F(X)$ being the 3485-th cyclotomic polynomial. In this situation the polynomial $F(X)$ has degree $N = \varphi(3485) = 2560$, and modulo two it factors into 64 polynomials each of degree 40. This polynomial therefore allows us to compute in parallel with up to 64 elements of any subfield of $\mathbb{F}_{2^{40}}$. For instance, by selecting $n = 1$ and $l = 64$ we perform 64 operations in $\mathbb{F}_2$ in parallel; selecting $n = 40$ and $l = 1$ we perform operations in a single copy of the finite field $\mathbb{F}_{2^{40}}$; whereas selecting $n = 8$ and $l = 16$ we perform SIMD operations on what is essentially the AES state matrix, namely 16 elements of $\mathbb{F}_{2^8}$.

3 Somewhat Homomorphic Scheme Supporting SIMD Operations in $\mathbb{K}_n$

In this section, we recall the Smart–Vercauteren variant of Gentry's somewhat homomorphic scheme and show that it can support SIMD operations in r copies of the finite field $\mathbb{K}_n$ by modifying key generation. Note that the recent FHE schemes based on ring-LWE [3] also support such style operations, and may be preferable in practice due to their improved key generation procedures; for an extension of some of the ideas in this paper to the ring-LWE schemes see [2, 13–16]. However, whilst our SIMD style operations extend to the ring-LWE based somewhat homomorphic schemes, our parallel recryption step does not carry over. We will return to this point later on.

3.1 Smart-Vercauteren somewhat homomorphic scheme

Let $F \in \mathbb{Z}[X]$ be a monic irreducible polynomial of degree N and let $\mathbb{K} = \mathbb{Q}(\theta) = \mathbb{Q}[X]/(F)$ denote the number field defined by F. Gentry's original scheme uses two co-prime ideals I and J in the number ring $\mathbb{Z}[\theta]$. The ideal I is chosen to have small norm $\mathcal{N}(I) = \sharp(\mathbb{Z}[\theta]/I)$ and determines the plaintext space, namely $\mathbb{Z}[\theta]/I$. For this reason, $I = (2)$ is chosen in practice. Note that in the case of a general F the quotient ring $\mathbb{Z}[\theta]/(2)$ is an algebra of a somewhat more general type than discussed in Section 2. We shall choose F later on such that one obtains precisely the type of algebra considered in Section 2. The ideal J determines the private/public key pair: the private key consists of a "good" representation of J, whereas the public key consists of a "bad" representation of J.

To clarify the notions of "good" and "bad", we first describe the Smart–Vercauteren instantiation. The ideal J is chosen to be principal, i.e. generated by one element $\gamma \in \mathbb{Z}[\theta]$, and has the following additional property: let $d = \mathcal{N}(J) = \sharp(\mathbb{Z}[\theta]/J) = |N_{\mathbb{K}/\mathbb{Q}}(\gamma)|$, where $N_{\mathbb{K}/\mathbb{Q}}(\cdot)$ denotes the number field norm of $\mathbb{K}$ to $\mathbb{Q}$, then there exists a unique $\alpha \in \mathbb{Z}_d$ such that

$$J = (\gamma) = (d, \theta - \alpha)\,.$$

The element α, and the integer d, can be computed in polynomial time by, for example, computing the Hermite Normal Form representation of the ideal.

The "good" representation of J (i.e. the private key) corresponds to the small generator γ, whereas the "bad" representation (i.e. public key) is $(d, \theta - \alpha)$. The additional property of J is equivalent with the requirement that the Hermite Normal Form representation of J has the following specific form

$$\begin{pmatrix} d & 0 & 0 \ldots 0 \\ -\alpha & 1 & 0 \quad 0 \\ -\alpha^2 & 0 & 1 \quad 0 \\ \vdots & & \ddots \\ -\alpha^{N-1} & 0 & 0 \quad 1 \end{pmatrix},$$

where the entries below d in the first column are taken modulo d. Another characterisation of this property is that the ideal J simply contains an element of the form $\theta - \alpha$. This is clearly necessary since J can be generated by $(d, \theta - \alpha)$, but it is also sufficient. Indeed, since $\gamma \in J$, this implies that $d \in J$, so $(d, \theta - \alpha) \subset J$ and since both ideals have the same norm, we must have $J = (d, \theta - \alpha)$. As such, there exists an element $\nu \in \mathbb{Z}[\theta]$ with $\nu \cdot \gamma = \theta - \alpha$. To derive an easy verifiable condition on γ, we define the algebraic number $\zeta \in \mathbb{Z}[\theta]$ such that

$$\zeta \cdot \gamma = d\,. \tag{1}$$

Multiplying $\nu \cdot \gamma = \theta - \alpha$ on both sides with ζ gives the condition $d \cdot \nu = \theta \cdot \zeta - \alpha \cdot \zeta$. Write $\zeta = \sum_{i=0}^{N-1} \zeta_i \cdot \theta^i$ and $F(X) = \sum_{i=0}^{N} F_i \cdot X^i$, then computing the product $\theta \cdot \zeta$ explicitly and reducing modulo d finally leads to:

$$\alpha \cdot \zeta_i = \zeta_{i-1} - \zeta_{N-1} F_i \bmod d\,, \tag{2}$$

for all $i = 0, \ldots, N-1$ where $\zeta_{-1} = 0$.

Note that the two element representation $(d, \theta - \alpha)$ defines an easily computable homomorphism

$$H : \mathbb{Z}[\theta] \to \mathbb{Z}_d : \eta = \sum_{i=0}^{N-1} \eta_i \cdot \theta^i \mapsto H(\eta) = \sum_{i=0}^{N-1} \eta_i \cdot \alpha^i \bmod d\,. \tag{3}$$

The homomorphism H also makes it very easy to test if an element $\eta \in \mathbb{Z}[\theta]$ is contained in the ideal J, namely $\eta \in J$ if and only if $H(\eta) = 0$. Furthermore, given the "good" representation γ, it is possible to invert H on a small subset of $\mathbb{Z}[\theta]$ as shown by the following lemma.

Lemma 1 *Let $J = (\gamma) = (d, \theta - \alpha)$ and $\zeta \cdot \gamma = d$ and let H be defined as in* (3)*. Let $\eta \in \mathbb{Z}[\theta]$ with $\|\eta\|_\infty < U$, then we have*

$$\eta = H(\eta) - \left\lceil \frac{H(\eta) \cdot \zeta}{d} \right\rfloor \cdot \gamma \qquad \text{for} \qquad U = \frac{d}{2 \cdot \delta_\infty \cdot \|\zeta\|_\infty},$$

where $\delta_\infty = \sup \left\{ \frac{\|\mu \cdot \nu\|_\infty}{\|\mu\|_\infty \cdot \|\nu\|_\infty} : \mu, \nu \in \mathbb{Z}[\theta] \right\}$. Furthermore, for $\|\eta\|_\infty < U$ we have

$$[H(\eta) \cdot \zeta]_d = [\eta \cdot \zeta]_d = \eta \cdot \zeta \,. \tag{4}$$

Proof It is easy to see that $H(\eta) - \eta$ is contained in the principal ideal generated by γ. As such, there exists a $\beta \in \mathbb{Z}[\theta]$ such that $H(\eta) - \eta = \beta \cdot \gamma$. Using $\zeta = d/\gamma$, we can write

$$\beta = \frac{H(\eta) \cdot \zeta}{d} - \frac{\eta \cdot \zeta}{d} \,. \tag{5}$$

Since β has integer coefficients, we can recover it by rounding the coefficients of the first term if the coefficients of the second term are strictly bounded by $1/2$. This shows that η can be recovered from $H(\eta)$ for $\|\eta\|_\infty < d/(2 \cdot \delta_\infty \cdot \|\zeta\|_\infty)$. Furthermore, equation (5) shows that $[H(\eta) \cdot \zeta]_d = [\eta \cdot \zeta]_d$ and since $\|\eta\|_\infty < U$, we have $[\eta \cdot \zeta]_d = \eta \cdot \zeta$.

Corollary 1 *Using the notation of Lemma 1, assume that $\|\eta\|_\infty < U/L$, for some $L > 1$, then for $i = 0, \ldots, N-1$ we have*

$$-\frac{1}{2L} < \frac{H(\eta) \cdot \zeta_i}{d} - \left\lceil \frac{H(\eta) \cdot \zeta_i}{d} \right\rfloor < \frac{1}{2L},$$

i.e. $H(\eta) \cdot \zeta_i / d$ is within distance $1/2L$ of an integer, where (as before) ζ_i is the ith coefficient of ζ in the polynomial basis.

Proof Follows directly from equation (5) and the assumption on η.

The above lemma shows that we can recover an element η from its image under H, when its norm is not too large. As such we obtain a trapdoor one way function that can be used as the basis for encryption. Using these preliminaries we are now ready to define key generation, encryption and decryption.

KEY GENERATION: Input parameters: N, t
Generate a monic irreducible polynomial $F \in \mathbb{Z}[X]$ of degree N with small coefficients, defining the number field $\mathbb{K} = \mathbb{Q}(\theta) = \mathbb{Q}[X]/(F)$. Choose an element $\gamma \in \mathbb{Z}[\theta]$ with $\gamma = 1 \bmod 2$ such that the coefficients of γ are smaller in absolute value than 2^t (at least one coefficient should be a t-bit integer). This can be done for example by uniformly selecting γ from all polynomials of degree $N-1$ with coefficients bounded by 2^t in absolute value, although other distributions are possible. Compute the norm $d = |N_{\mathbb{K}/\mathbb{Q}}(\gamma)|$ as well as the element $\zeta \in \mathbb{Z}[\theta]$ with $\zeta \cdot \gamma = d$. If d is even, choose a new γ. If d is odd, compute $\alpha = -\zeta_{N-1} \cdot F_0 / \zeta_0$ and verify whether (2) holds for all $i = 1, \ldots, N-1$. If not, generate a new γ. Otherwise, the public key is the pair $\mathsf{pk} := (d, \alpha)$ whereas the private key is the element $\mathsf{sk} := \zeta$.

In practice, N will be of the order a few thousand and t a few hundred. The size of d can be approximated roughly by $N^N \cdot 2^{Nt}$; this therefore results in a d of several million bits.

ENCRYPTION: Input parameters: μ, $\mathsf{pk} := (d, \alpha)$, message $M \in A := \mathbb{F}_2[X]/(F(X))$
The plaintext space consists of (a subalgebra of) the algebra $A := \mathbb{F}_2[X]/(F(X))$. Represent the message M as a polynomial $M(X) \in \mathbb{Z}[X]$ with coefficients in $\{0, 1\}$. Uniformly generate a "noise" polynomial $R(X) \in \mathbb{Z}[X]$ of degree $< N$, subject to with $\|R(X)\|_\infty \leq \mu$, and compute the ciphertext as

$$c \leftarrow [M(\alpha) + 2 \cdot R(\alpha)]_d \,.$$

Note that the ciphertext is an element in $\mathbb{Z}_d$ and that encryption simply corresponds to applying the homomorphism H to the algebraic integer $C(\theta) := M(\theta) + 2 \cdot R(\theta)$. Furthermore, it should be clear that if we can recover $C(\theta)$, then we can decrypt simply by computing $C(X) \bmod 2$. The encryption function is denoted as $c \leftarrow \mathsf{Encrypt}(M(X), \mathsf{pk})$. If $M(X) \in A$ then we say $M\big|_\alpha = M(\alpha) \pmod d$ is a "trivial" encryption of $M(X)$, i.e. it is an encryption with no randomness.

DECRYPTION: Input parameters: ciphertext $c \in \mathbb{Z}_d$, $\mathsf{sk} := \zeta$
Given the ciphertext $c \in \mathbb{Z}_d$, compute the element $C(\theta)$ as

$$C(\theta) = c - \left\lceil \frac{c \cdot \zeta}{d} \right\rfloor ,$$

and then set $M(X) = C(X) \bmod 2$. Note that here we used the fact that $\gamma \equiv 1 \bmod 2$. We can obtain a simpler decryption procedure using the last statement in Lemma 1. Indeed, if c is a decryptable ciphertext, we know that $\|C(\theta)\|_\infty < U$ and thus that

$$[c \cdot \zeta]_d = C(\theta) \cdot \zeta \,.$$

Since $\gamma \equiv 1 \bmod 2$ and d is odd with $d = \gamma \cdot \zeta$, we see that also $\zeta \equiv 1 \bmod 2$. Furthermore, $C(\theta) = M(\theta) + 2R(\theta)$, so we obtain

$$[c \cdot \zeta]_d \bmod 2 = M(\theta) \bmod 2 = M(X)\,.$$

This shows that for $\zeta = \sum_{i=0}^{N-1} \zeta_i \theta^i$ we can recover the coefficients of $M(X) = m_0 + m_1 \cdot X + \cdots + m_{N-1} \cdot X^{N-1}$ one by one, by computing

$$m_i = [c \cdot \zeta_i]_d \pmod 2\,.$$

We write $M(X) \leftarrow \mathsf{Decrypt}(c, \mathsf{sk})$. Note that to save space for key storage, it suffices to store ζ_0, since the other ζ_i follow from equation (2). In particular, we obtain the closed expression $\zeta_i = w_i \cdot \zeta_0$ with

$$w_i = -\frac{1}{F_0}\left(\sum_{j=i+1}^{N} F_j \cdot \alpha^{j-i}\right) \pmod d\,. \tag{6}$$

Since the w_i can be publicly computed, we can decrypt $m_i = [c \cdot w_i \cdot \zeta_0]_d \pmod 2$. We pause to note that it is this linear relationship between the distinct decryption keys ζ_i which enables the parallel recryption procedure we describe later. For ring-LWE based somewhat homomorphic schemes supporting SIMD operations, where such a simple linear relation does not hold, it seems much harder to produce a parallel recryption procedure using the squashing paradigm of Gentry. Although see [14] for a possibly more efficient method in this direction.

HOMOMORPHIC OPERATIONS: It is easy to see that the scheme is somewhat homomorphic, where the operations being performed are addition and multiplication of ciphertexts modulo d. Indeed, let $c_i = H(C_i(\theta)) = H(M_i(\theta) + 2R_1(\theta))$ for $i = 1, 2$, then we have that

$$\begin{aligned} c_1 + c_2 &= H(M_1(\theta) + M_2(\theta) + 2(R_1(\theta) + R_2(\theta))) \\ c_1 \cdot c_2 &= H(M_1(\theta) \cdot M_2(\theta) + 2(M_1(\theta)R_2(\theta) + M_2(\theta)R_1(\theta) + 2R_1(\theta)R_2(\theta)))\,. \end{aligned}$$

This shows that operations on the ciphertext space induce corresponding operations on the plaintext space, i.e. the algebra A. Thus it is clear that the somewhat homomorphic scheme supports SIMD operations and operations on elements in possibly large degree (i.e. degree n) finite fields. To make a distinction when we are performing homomorphic operations we will use the notation $\oplus$ and $\odot$ to denote the homomorphic addition and multiplication of ciphertexts.

3.2 Efficient key generation and SIMD operations

Whilst the FHE scheme works for any polynomial F with small coefficients, the common case, as in [12] and [23], is to use the polynomial $F(X) := X^{2^n} + 1$. As pointed out by Gentry and Halevi [12] this enables major improvements in the key generation procedure over that proposed by Smart and Vercauteren [23]. If we let η_i denote the roots of the polynomial F over the complex numbers, or over a sufficiently large finite field, then we can compute ζ and d as follows:

- Compute $\omega_i \leftarrow \gamma(\eta_i) \in \mathbb{C}$ for all i.
- Compute $d \leftarrow \prod \omega_i$.
- Compute $\omega_i^* \leftarrow 1/\omega_i$.
- Interpolate the polynomial ζ/d from the data values ω_i^*.

The key observation is that since $F(X)$ is of the form $X^{2^n} + 1$, the η_i are 2^{n+1}-th roots of unity and so to perform the polynomial evaluation and interpolation above we can apply the Fast Fourier Transform (FFT). Indeed, Gentry and Halevi present an even more optimized scheme to compute d and ζ which requires only polynomial arithmetic, but this makes significant use of the fact that the trace of 2-power roots of unity is always zero.

The problem with selecting $F(X) = X^{2^n} + 1$ is that it has only one irreducible factor modulo two. In particular if we select $F(X) = X^{2^n} + 1$ then the underlying plaintext algebra is given by

$$A := \mathbb{F}_2[X]/(F) \cong \mathbb{F}_2[X]/(X-1)^{2^n}\,.$$

In other words, F does not split into a set of *distinct* irreducible factors modulo two as we required to enable SIMD operations.

We now present a possible replacement for $F(X)$. The key observation is that we need an $F(X)$ which enables fast key generation via FFT like algorithms, which has small coefficients, and which splits into distinct irreducible factors modulo two of the same degree. In addition we need a relatively large supply of such polynomials to cope with increasing security levels (i.e. N), different numbers of parallel operations (i.e. l) and different degree two finite fields in which operations occur (i.e. n). In particular need to pick an $F(X)$ which generates a Galois extension of degree n. In addition we need to select a polynomial $F(X)$ such that 2 is neither ramified, nor an index divisor, in the associated

number field generated by a root of $F(X)$. These conditions ensure that the algebra mod two splits into distinct finite fields of the same degree.

One is then led to consider other cyclotomic polynomials as follows. We select an odd integer m and recall that the m-th cyclotomic polynomial is defined by

$$\Phi_m(X) := \prod_{\eta}(X - \eta)$$

where η ranges over all m-th primitive roots of unity. We have $\deg(\Phi_m(X)) = \phi(m)$, and that $\Phi_m(X)$ is an irreducible polynomial with integer coefficients. In the practical range for m, the coefficients of Φ_m are very small, e.g. for all $m \leq 40000$ the coefficients are bounded by 59 and are in most cases much smaller than this upper bound.

The field $\mathbb{Q}(\theta)$ is a Galois extension and hence each prime ideal splits in $\mathbb{Q}(\theta)$ into a product of prime ideals of the same degree and ramification index. If m is odd then the prime two does not ramify in the field $\mathbb{Q}(\theta)$, nor is it an index divisor. In particular, by Dedekind's criterion, this means that the polynomial $\Phi_m(X)$, of degree $N = \phi(m)$, factors modulo two into a product of $r = N/d$ distinct irreducible polynomials of degree equal to the unique degree d of the prime ideals lying above the ideal (2). This degree d is the smallest integer such that $2^d \equiv 1 \pmod{m}$.

Hence, by selecting $F(X) := \Phi_m(X)$ in our construction of the algebra A over $\mathbb{F}_2$, we find that A is isomorphic to a product of r finite fields of degree $d = N/r$. The only issue is whether one can perform the key generation efficiently. To do this we use Fourier Transforms with respect to the m-th roots of unity. In particular given the polynomial γ in the key generation procedure we compute the evaluation at the m-th roots of unity via a Fourier Transform, and produce the norm d by selecting the N required values to multiply together (consisting of the evaluations of the primitive roots of unity). One can then compute $1/\gamma$ by inverting the Fourier coefficients and then interpolating via the inverse Fourier Transform.

In other words the same optimization as mentioned earlier can be applied: Instead of taking the standard Cooley-Tukey [7] FFT method for powers of two, we apply the Good-Thomas method [17,25] for when m is a product of two coprime integers, or Cooley-Tukey when m is a prime power. Either method reduces the problem to computing FFTs for prime power values of m, for which we can use the Rader FFT algorithm [21]. This in itself reduces the problem to computing a convolution of two sequences, which is then performed by extension of the sequences to length a power of two followed by the application of the Cooley-Tukey algorithm to the extended sequence. Overall the FFT then takes $O(m \cdot \log m)$ operations on elements of size $O(\log_2 d)$ bits. In practice $m \approx 2 \cdot N$ and so this gives the same complexity for key generation as using $F(X) = X^{2^n} + 1$, however the implied constants are slightly greater. This means we can achieve almost the same complexity for key generation as in the 2-power root of unity case. In [22] the above approach is extended and further optimizations are applied, so as to reduce the cost to nearer to what one sees when using $F(X) = X^{2^n} + 1$.

4 Fully Homomorphic Scheme and Naive Recryption Method

To turn the somewhat homomorphic scheme of the previous section into a fully homomorphic scheme, we follow Gentry's bootstrapping approach, i.e. we squash the decryption circuit so much that it can be evaluated by the somewhat homomorphic scheme. In particular, we use the optimized procedure described by Gentry and Halevi in [12].

4.1 The Recryption Method of Gentry and Halevi

Recall that each message bit m_i can be recovered as $m_i = [c \cdot w_i \cdot \zeta_0]_d \pmod 2$ with the w_i being publicly computable constants defined in (6). Since $[c \cdot w_i]_d$ can be computed without knowledge of ζ_0 it suffices to show how $[c \cdot \zeta_0]_d \pmod 2$ can be computed with a low complexity circuit.

The idea is to write the private key ζ_0 as the solution to a sparse-subset-sum problem. In particular, we will define s sets of S elements as follows (a discussion on the sizes of s and S will be given later): choose s elements $x_i \in [0, \ldots, d)$, a random integer $R \in [1, \ldots, d)$ and define the i-th set $\mathcal{B}_i = \{x_i \cdot R^j \pmod d \mid j \in [0, \ldots, S)\}$ such that the private key ζ_0 can be written as the sum

$$\zeta_0 = \sum_{i=1}^{s} \sum_{j=0}^{S-1} b_{i,j} \cdot x_i \cdot R^j \pmod d,$$

where for each i only one $b_{i,j} = 1$ and all other $b_{i,j}$ are zero. The index j for which $b_{i,j} = 1$ will be denoted by e_i and so we can write $\zeta_0 = \sum_{i=1}^{s} x_i \cdot R^{e_i} \pmod d$. The result is that we have written ζ_0 as the sum of s elements, where one element is taken from each $\mathcal{B}_i$. To enable recryption or ciphertext cleaning, we will augment the public key with

additional information: compute the ciphertexts $c_{i,j} \leftarrow \mathsf{Encrypt}(b_{i,j}, \mathsf{pk})$ for $1 \le i \le s,\ 0 \le j < S$, then the public key now consists of the data

$$\left(d, \alpha, s, S, R, \left\{x_i, \{c_{i,j}\}_{j=0}^{S-1}\right\}_{i=1}^{s}\right).$$

Denote $y_{i,j} = c \cdot x_i \cdot R^j \pmod d$ for $i = 1, \ldots, s$ and $j = 0, \ldots, S-1$ such that $0 \le y_{i,j} < d$, then the decryption function $[c \cdot \zeta_0]_d \pmod 2$ can be rewritten as

$$\begin{aligned}
[c \cdot \zeta_0]_d \pmod 2 &= \left[\sum_{i=1}^{s}\sum_{j=0}^{S-1} b_{i,j} \cdot y_{i,j}\right]_d \pmod 2 \\
&= \left(\sum_{i=1}^{s}\sum_{j=0}^{S-1} b_{i,j} \cdot y_{i,j}\right) - d \cdot \left\lfloor \sum_{i=1}^{s}\sum_{j=0}^{S-1} b_{i,j} \cdot \frac{y_{i,j}}{d} \right\rfloor \pmod 2 \\
&= \bigoplus_{i=1}^{s}\bigoplus_{j=0}^{S-1} b_{i,j} \cdot y_{i,j} \pmod 2 \oplus \left\lfloor \sum_{i=1}^{s}\sum_{j=0}^{S-1} b_{i,j} \cdot \frac{y_{i,j}}{d} \right\rfloor \pmod 2 .
\end{aligned}$$

Note that the latter double sum $\mathcal{T} = \sum_{i=1}^{s}\sum_{j=0}^{S-1} b_{i,j} \cdot \frac{y_{i,j}}{d}$ is equal to $c \cdot \zeta_0/d$ and if we assume that c is the image of $C(\theta)$ under H, where $\|C(\theta)\|_\infty < U/(s+1)$, then we know by Corollary 1 that $\mathcal{T}$ is within distance $1/2(s+1)$ of an integer. If we now replace each $\frac{y_{i,j}}{d}$ with an approximation $z_{i,j}$ up to p bits after the binary point, i.e. $|z_{i,j} - y_{i,j}/d| < 2^{-(p+1)}$, then since there are only s non-zero terms, we have that $|\mathcal{T} - \sum_{i=1}^{s}\sum_{j=0}^{S-1} b_{i,j} \cdot z_{i,j}| < s \cdot 2^{-(p+1)}$. Rounding the double sum over the $z_{i,j}$ will thus give the same result as rounding $\mathcal{T}$ as long as

$$\frac{1}{2(s+1)} + s \cdot 2^{-(p+1)} < 1/2 ,$$

which implies that $p \ge \lceil \log_2(s+1) \rceil$. Furthermore, in the inner sum we are adding S numbers of which only one is non-zero. As such, we can compute the k-th bit of this sum by simply XOR-ing the k-th bits of the $b_{i,j} \cdot z_{i,j}$ for $j = 1, \ldots, S$. We are then left with an addition of s numbers, each which consists of p bits after the binary point.

We are now ready to formulate the recrypt algorithm by mapping these equations into the encrypted domain. To this end, we require two helper functions. The first function $\mathbf{b} \leftarrow \mathsf{compute\_bits}(y)$ takes as input an integer $0 \le y < d$ and outputs the vector of bits $\mathbf{b} = (b_0, b_1, \ldots, b_p)$ such that

$$\left| \frac{y}{d} - (b_0 + \frac{b_1}{2} + \frac{b_2}{2^2} + \cdots + \frac{b_p}{2^p}) \right| < \frac{1}{2^{p+1}}.$$

This is easily computed by determining $u \leftarrow \lceil (2^p \cdot y)/d \rfloor$, and then reading the bits from the (small) integer u.

The second function $\mathsf{school\_book\_add}(A)$ takes as input an $s \times (p+1)$ array A of ciphertexts, where each row contains the encryptions of the $(p+1)$ bits of an integer. The result of the function is a $(p+1)$ vector containing the encryptions of the $(p+1)$ bits of the sum of these s integers modulo 2^{p+1}. The school book method is discussed in more detail in [12] where it is shown that it takes time

$$T_{\mathsf{school\_book\_add}} := \left(s \cdot 2^{p-1} + \sum_{k=1}^{p-1} (s+k) \cdot 2^{p-k} \right) \cdot T_{\mathsf{mod},d}$$

where $T_{\mathsf{mod},d}$ denotes the time of performing one multiplication modulo d.

In Algorithm 1 we present the algorithm for recrypting the first bit of the message underlying a ciphertext c, i.e. the algorithm computes $[c \cdot \zeta_0]_d \pmod 2$ in the encrypted domain using the augmented public key. This is essentially the recryption algorithm used by Gentry and Halevi, where the message space is one bit only.

4.2 Some Initial Modifications

Before progressing to our parallel recryption method we first pause to re-examine the Gentry-Halevi method in the context of largest message spaces. To obtain the recyption of the i-th coefficient we simly input $[c \cdot w_i]_d$ instead of c, since decrypting the i-th bit is given by $[c \cdot w_i \cdot \zeta_0]_d \pmod 2$. We denote the cost of executing this algorithm for a one bit ciphertext as T_{bits}. Ignoring the modular additions, we see that $T_{\mathsf{bits}} = \left((S+1) \cdot s \cdot + s \cdot 2^{p-1} + \sum_{k=1}^{p-1}(s+k) \cdot 2^{p-k}\right) \cdot T_{\mathsf{mod},d}$.

Algorithm 1: BitRecrypt(c, pk): Recrypting the First Bit of the Plaintext Associated With Ciphertext c

$A \leftarrow 0$, where $A \in M_{s\times(p+1)}(\mathbb{Z}_d)$.
sum $\leftarrow 0$.
for i **from** 1 **upto** s **do**
  $y \leftarrow c \cdot x_i \pmod d$.
  for j **from** 0 **upto** $S-1$ **do**
    if y is odd **then**
      sum $\leftarrow$ sum $\oplus\, c_{i,j}$.
    $\mathbf{b} \leftarrow \mathsf{compute\_bits}(y)$.
    for u **from** 0 **upto** p **do**
      $A_{i,u} \leftarrow A_{i,u} \oplus (\mathbf{b}_u \cdot c_{i,j})$.
    $y \leftarrow y \cdot R \pmod d$.
$\mathbf{a} \leftarrow \mathsf{school\_book\_add}(A)$.
$\overline{c} \leftarrow$ sum $\oplus\, \mathbf{a}_0$.
return $(\overline{c})$.

To recrypt a whole ciphertext c, we first form ciphertexts $\overline{c}_i = \mathsf{BitRecrypt}([c\cdot w_i]_d, \mathsf{pk})$ for $i = 0, \ldots, N-1$, which are recryptions of the coefficients of the underlying polynomial $M(X)$ by submitting $[c \cdot w_i]_d$ to Algorithm 1. Then given $\overline{c}_i$ we form the ciphertext

$$\overline{c} \leftarrow \sum_{i=0}^{N-1} \overline{c}_i \odot \alpha^i$$

which will be a recryption of the original ciphertext. Note, to control the noise this last sum is computed naively, and not via Horner's rule, i.e. we multiply each coefficient ciphertext $\overline{c}_i$ by $\alpha^i \pmod d$ and then sum. The resulting algorithm is summarized in Algorithm 2. Assuming the $\alpha^i \pmod d$ and w_i are precomputed, the total cost of recrypting a ciphertext

Algorithm 2: Recrypting Ciphertext c version 1

$\overline{c} \leftarrow 0$.
for i **from** 0 **upto** $N-1$ **do**
  $\overline{c}_i \leftarrow \mathsf{BitRecrypt}([c\cdot w_i]_d, \mathsf{pk})$.
  $\overline{c} \leftarrow \overline{c} \oplus \overline{c}_i \odot \alpha^i$.
return $(\overline{c})$.

corresponding to an arbitrary element in A (using our naive method) is essentially $N \cdot T_{\mathsf{bits}} + 2 \cdot N \cdot T_{\mathsf{mod},d}$. If SIMD style operations, and operations on larger datatypes, are to be supported we therefore need a more efficient method to perform recryption; since the above cost could be prohibitive. We therefore now turn to utilizing our SIMD operations to improve the performance of recryption of such ciphertexts.

5 Parallel Recryption

Whilst Algorithm 1 will recrypt a ciphertext that encodes an element of the algebra A, it can be made significantly more efficient. Firstly, the procedure recrypts a general element in A, yet in practice we will only have that c contains $l\cdot n \leq N$ encrypted bits. Secondly, since the recrypt procedure is a binary circuit we can run it on the r embedded copies of $\mathbb{F}_2$, i.e. we can use the SIMD style operations to recrypt r bits in parallel.

The first optimization is easy to obtain: recall that $\Gamma_{n,l}$ maps a vector of l binary polynomials $(\kappa_1(\psi), \ldots, \kappa_l(\psi))$ each of of degree less than n, into a polynomial $a(X)$ of degree less than N. The map $\Gamma_{n,l}$ thus defines an isomorphism between $\mathbb{K}_n^l$ and $\Gamma_{n,l}(\mathbb{K}_n^l)$ so $\Gamma_{n,l}^{-1}$ is well defined on the result of the computation. We can represent $\Gamma_{n,l}^{-1}$ explicitly by an $(n\cdot l)\times N$ binary matrix B over $\mathbb{F}_2$ which is defined as follows:

$$\mathsf{coeff}(\kappa_i, j) = \sum_{k=0}^{N-1} B_{j+i\cdot n+1,k+1} \cdot \mathsf{coeff}(a(X), k).$$

Using B we can therefore first obtain encryptions of all the coefficients of the κ_i, recrypt these using Algorithm 1 and then reconstruct the recrypted ciphertext using $\Gamma_{n,l}$. In particular, denote with $\overline{c}_{i_1,i_2}$ a recryption of the i_1th coefficient of the i_2th component in $\mathbb{K}_n^l$, then we can obtain a full recryption of an element in $\mathbb{K}_n^l$ by computing

$$\overline{c} \leftarrow \sum_{i_1=0}^{n-1} \sum_{i_2=1}^{l} \overline{c}_{i_1,i_2} \odot \left(\left(\Gamma_{n,l}(0,\ldots,0,\psi^{i_1},0,\ldots,0) \right) \big|_\alpha \right),$$

where $(0,\ldots,0,\psi^{i_1},0,\ldots,0) \in \mathbb{K}_n^l$ is the element whose i_2th component is equal to ψ^{i_1}, and $M(X)\big|_\alpha$ is the trivial encryption of the element $M(X)$ in the algebra A.

Recall that given a ciphertext c, the value $[c \cdot w_i]_d$ is an encryption of the ith coefficient of $a(X)$. Since the scheme is homomorphic and using the matrix B we conclude that

$$c_{i_1,i_2} = \left[\sum_{k=0}^{N-1} B_{i_1+i_2\cdot n+1,k+1}[c\cdot w_k]_d\right]_d = \left[c\cdot\left(\sum_{k=0}^{N-1} B_{i_1+i_2\cdot n+1,k+1}\cdot w_k\right)\right]_d$$

is a valid encryption of $\mathsf{coeff}(\kappa_{i_2}, i_1)$. Note that these quantities are obtained as the sum of maximum N ciphertexts, which implies that the original c has to be an encryption of $C(\theta)$ with $\|C(\theta)\|_\infty < U/((s+1)\cdot N)$ for Algorithm 1 to recrypt correctly. The second algorithm thus first computes the $n \cdot l$ constants (the w_i are no longer required)

$$v_{i_1,i_2} = \sum_{k=0}^{N-1} B_{i_1+i_2\cdot n+1,k+1}\cdot w_k \pmod d,$$

and then computes the recryptions $\overline{c}_{i_1,i_2} = \mathsf{BitRecrypt}([c\cdot v_{i_1,i2}]_d, \mathsf{pk})$. Notice how we have reduced the number of calls to recrypt from N down to $n \cdot l$ and that we require only $n \cdot l$ constants v_{i_1,i_2} instead of the N constants w_i. The result is summarized in Algorithm 3. Assuming the $\left(\Gamma_{n,l}(0,\ldots,0,\psi^{i_1},0,\ldots,0)\right)\big|_\alpha$ and v_{i_1,i_2} are precomputed, the total cost of recrypting a ciphertext is essentially $n\cdot l\cdot T_{\mathsf{bits}} + 2\cdot n\cdot l\cdot T_{\mathsf{mod},d}$.

Algorithm 3: Recrypting Ciphertext c version 2

$\overline{c} \leftarrow 0$.
for i_1 **from** 0 **upto** $n-1$ **do**
  for i_2 **from** 0 **upto** $l-1$ **do**
    $\overline{c}_{i_1,i_2} \leftarrow \mathsf{BitRecrypt}([c\cdot v_{i_1,i_2}]_d, \mathsf{pk})$.
    $\overline{c} \leftarrow \overline{c} \oplus \overline{c}_{i_1,i_2} \odot \left(\Gamma_{n,l}(0,\ldots,0,\psi^{i_1},0,\ldots,0)\right)\big|_\alpha$.
return $(\overline{c})$.

So far we have not exploited the SIMD capabilities of the somewhat homomorphic scheme. Therefore our next goal is to produce the recryptions $\overline{c}_{i_1,i_2}$ in parallel for $i_2 = 1,\ldots,l$. Thus we aim to compute a ciphertext $\hat{c}_{i_1}$ from c such that $\hat{c}_{i_1}$ represents a recryption of the message

$$(\mathsf{coeff}(\kappa_1, i_1),\ldots,\mathsf{coeff}(\kappa_l, i_1)),$$

where c represents an encryption of $(\kappa_1,\ldots,\kappa_l)$. We use the notation $\hat{c}_i$ to distinguish it from the recryption $\overline{c}_i$ above.

The key observation is that the recrypt procedure is the evaluation of a binary circuit, and that this binary circuit is identical (bar the constants) no matter which component we are recrypting. In addition the algebra splits into (at least) l finite fields of characteristic two, thus we can embed the binary circuit into each of these l components and perform the associated recryption in parallel. For a fixed i_1 we therefore want to execute the computation of the vector

$$([c\cdot v_{i_1,1}\cdot\zeta_0]_d \pmod 2,\ldots,[c\cdot v_{i_1,l}\cdot\zeta_0]_d \pmod 2)$$

in the encrypted domain in parallel. Recall that each component of this vector is computed as

$$[c\cdot v_{i_1,k}\cdot\zeta_0]_d \pmod 2 = \bigoplus_{i=1}^{s}\bigoplus_{j=0}^{S-1} b_{i,j}\cdot y_{i,j}^{(k)} \pmod 2 \oplus \left\lfloor\sum_{i=1}^{s}\sum_{j=0}^{S-1} b_{i,j}\cdot z_{i,j}^{(k)}\right\rfloor \pmod 2,$$

where $y_{i,j}^{(k)} = c\cdot v_{i_1,k}\cdot x_i\cdot R^j$ and $z_{i,j}^{(k)}$ an approximation of $y_{i,j}^{(k)}/d$ up to p bits after the binary point. Recall that to obtain the bit $\mathcal{B}_k = \left\lfloor\sum_{i=1}^{s}\sum_{j=0}^{S-1} b_{i,j}\cdot z_{i,j}^{(k)}\right\rfloor \pmod 2$ we used the function $\mathsf{school\_book\_add}(M)$ with input an $s\times(p+1)$ array M where the ith row contained $\oplus_{j=0}^{S-1} b_{i,j}\cdot\mathsf{compute\_bits}(y_{i,j}^{(k)})$. In fact, $\mathcal{B}_k$ was simply the first bit in the bit vector returned by $\mathsf{school\_book\_add}(M)$.

If we now want to execute the above computation in the kth component (instead of the first), we basically have to multiply everything by $\Gamma_{n,l}(0,\ldots,0,1,0,\ldots,0)$, where $(0,\ldots,0,1,0,\ldots,0)$ is the vector of l elements of $\mathbb{K}_n$ whose kth element is equal to one, with all other elements being zero. To avoid costly modular multiplications by

$\Gamma_{n,l}(0,\ldots,0,1,0,\ldots,0)\big|_\alpha$, we will use l different encryptions of $b_{i,j}$, depending on which of the l components of the algebra we are using. In particular, we no longer augment the public key with the data

$$\left(p, s, S, R, \left\{x_i, \{c_{i,j}\}_{j=0}^{S-1}\right\}_{i=1}^{s}\right),$$

where $c_{i,j} \leftarrow \mathsf{Encrypt}(b_{i,j}, \mathsf{pk})$, but instead replace the $c_{i,j}$ components with elements $e_{i,j,k}$ where

$$e_{i,j,k} \leftarrow \mathsf{Encrypt}\left(b_{i,j} \cdot \Gamma_{n,l}(0,\ldots,0,1,0,\ldots,0), \mathsf{pk}\right) \text{ for } 1 \le i \le s,\ 0 \le j < S,\ 0 \le k < l\,.$$

This means we need to increase the size of the augmented public key by essentially a factor of l. Once we have computed all the $\hat{c}_{i_1}$'s we can simply recover $\overline{c}$ by computing

$$\overline{c} \leftarrow \sum_{i_1=0}^{n-1} \hat{c}_{i_1} \odot \left(\left(\Gamma_{n,l}(\psi^{i_1},\ldots,\psi^{i_1})\right)\big|_\alpha\right).$$

The resulting algorithm is given in Algorithm 4. Note that to compute each $\hat{c}_{i_1}$ we only require one call to the function $\mathsf{school\_book\_add}(A)$; compared to l calls in Algorithm 3.

Algorithm 4: Recrypting Ciphertext c version 3: parallel recryption of all i_1th coefficients of the n elements embedded in a ciphertext c

$\overline{c} \leftarrow 0$.
for i_1 **from** 0 **upto** $n-1$ **do**
  sum $\leftarrow 0$.
  $A \leftarrow 0$, where $A \in M_{s\times(p+1)}(\mathbb{Z}/d\mathbb{Z})$.
  for i_2 **from** 0 **upto** $l-1$ **do**
    $c_{i_1,i_2} \leftarrow c \cdot v_{i_1,i_2} \pmod d$.
    for j **from** 1 **upto** s **do**
      $y \leftarrow c_{i_1,i_2} \cdot x_j \pmod d$.
      for k **from** 0 **upto** $S-1$ **do**
        if y is odd **then**
          sum $\leftarrow$ sum $\oplus\, e_{j,k,i_2}$.
        $\mathbf{b} \leftarrow \mathsf{compute\_bits}(y)$.
        for u **from** 0 **upto** p **do**
          $A_{j,u} \leftarrow A_{j,u} \oplus (\mathbf{b}_u \cdot e_{j,k,i_2})$.
        $y \leftarrow y \cdot R \pmod d$.
  $\mathbf{a} \leftarrow \mathsf{school\_book\_add}(A)$.
  $\hat{c}_{i_1} \leftarrow$ sum $\oplus\, \mathbf{a}_0$.
  $\overline{c} \leftarrow \overline{c} \oplus \hat{c}_{i_1} \odot \left(\left(\Gamma_{n,l}(\psi^{i_1},\ldots,\psi^{i_1})\right)\big|_\alpha\right)$.
return $(\overline{c})$.

We let $T_{\mathsf{par}}(n,l)$ denote the cost of performing this recryption operation on a message consisting of l field elements from $\mathbb{K}_n$ held in parallel. Assuming the $\left(\Gamma_{n,l}(\psi^{i_1},\ldots,\psi^{i_1})\right)\big|_\alpha$ and the v_{i_1,i_2} are precomputed we obtain that

$$T_{\mathsf{par}}(n,l) = n\,(S\cdot s\cdot l + s\cdot l + l + 1)\cdot T_{\mathsf{mod},d} + n\cdot T_{\mathsf{school\_book\_add}}\,.$$

The main cost advantage therefore stems from the fewer calls to the function school_book_add.

Naively it would appear that our parallel version of recrypt, using Algorithm 4, is more efficient than the naive version using Algorithm 2. However, one may need larger public keys to actually implement the parallel recryption (as it is a more complex circuit). We also need to compare whether doing operations in parallel and with large data entries (via the algebra A) is more efficient than doing the same operations but with bits using the standard bit-wise FHE scheme but with more complex circuits. It is to this topic we now turn by examining some "toy" examples for small security parameters:

6 Security Analysis and Parameters

The analysis of Gentry of the basic FHE scheme and associated bootstrapping operation applies in our situation. The security of the underlying somewhat homomorphic scheme is based on the hardness of a variant of the bounded distance decoding (BDDP) problem; whereas the security of the bootstrapping procedure is based on the sparse subset sum problem (SSSP). Indeed the minor modifications we make in future sections to the public key result in exactly the same

security reductions. Thus an adversary against the scheme can either be turned into an algorithm to solve a decision variant of the BDDP, or a SSSP.

When selecting key sizes for cryptographic schemes, in practice one almost always selects key sizes based on the *best known attacks* and not on the hard problems from which a security problem reduces. We have various parameters we need to select s, S, N, t and μ. The sizes of N, t and μ determine whether one can break the scheme by distinguishing ciphertexts, or (more seriously) by message or key recovery. Parameter selection is here based on the hardness of solving explicit closest vector problems (CVPs), in lattices of dimension N, involving basis matrices with coefficients bounded by d (a function of t and N), and for close vectors whose distance to the lattice is related to the size of μ. An algorithm to solve the CVP/BDDP can be directly used to recover plaintexts as explained in [23]. The larger the ratio of t to μ the easier it is to recover plaintexts, but the ratio of t to μ also determines how complicated a circuit the basic somewhat homomorphic scheme can evaluate. Indeed the smaller the ratio of t to μ the less expressive our somewhat homomorphic scheme is. In selecting N, t and μ one needs to make a careful analysis of the current state of the art in lattice basis reduction; a topic which is beyond the scope of this paper.

On the other hand, it is not the case that an algorithm to solve the sparse subset sum problem can be used to break the scheme. The security proof in [11] uses the FHE adversary to solve the following SSSP

$$\zeta_0 = \sum_{i=1}^{s} \sum_{j=0}^{S-1} b_{i,j} \cdot (x_i \cdot R^j) \pmod{d}.$$

The simulator (solving SSSP) is given ζ_0 and the weights $x_i \cdot R^j \pmod{d}$, and uses random ciphertexts $c_{i,j}$ to represent the encryption of the $b_{i,j}$. Since the proof has already shown that ciphertexts of specific values are indistinguishable from encryptions of random values, the adversary does not know it is in a simulation. The proof in [11] shows how the simulator can then solve the SSSP. Whilst this easily establishes the fact that the recrypt procedure does not reduce the security of the scheme, assuming of course the scheme is KDM secure and the SSSP is hard, it actually tells us very little in practice. In particular it says: "If the adversary knows the secret key, then recovering another representation of the secret key is equivalent to solving the SSSP".

The parameters s and S determine (in practice) a hidden sparse subset sum problem rather than a standard SSSP. Namely, the adversary needs to solve the above subset sum problem where he is not given access to the value ζ_0. Taking the pragmatic view of parameter selection based on the best known attack, it is clear that neither the lattice attacks on the SSSP nor the time-memory trade off methods to solve the SSSP apply in the hidden case. This has important direct implications for parameter size selection. For example, if a time-memory trade off is possible then we need to select S and s such that $S^{\lfloor s/2 \rfloor} > 2^\lambda$, where we do not believe the adversary can perform 2^λ operations. However, since the time-memory trade off against the hidden SSSP appears impossible, we select can instead select $S^s > 2^\lambda$.

This observation has a number of consequences: Firstly we can select S to be much smaller than Gentry–Halevi do, secondly this means we do not need to complicate the recryption procedure with the index encoding method they use to save space, since S is now small enough to not require it. Thirdly this halves the degree of the resulting recryption circuit which makes the scheme more efficient, and fourthly it saves on the computational cost of recryption, since we need to do less work.

In summary: in practice one should select N, t and μ according to best practice from lattice basis reduction. For real systems this means that parameters need to be chosen that are significantly larger than the toy examples presented in Gentry–Halevi. However, when selecting s and S one can be less conservative than Gentry–Halevi.

In Section 5 we detailed a parallel recryption procedure which has the same multiplicative depth as the one above; but which requires more addition operations, where the number of extra additions depends on the level of SIMD operations required. Thus the value of t may need to be larger than that required in non SIMD based schemes. Asymptotically the constant increase will make no difference, but for "practical" parameters one may have a noticeable difference. Thus we now turn to presenting experimental results for "toy" security levels. This is done purely to show that our algorithms make a difference even for choices of N, μ and t corresponding to low security levels.

7 Experimental Results

So the question arises as to whether it is simpler to perform FHE on bits, or to perform FHE via the algebra A. In this section we concentrate on estimating the performance in terms of the run time and the sizes of the resulting ciphertexts which need to be stored. First recall key generation; we choose N and a polynomial $F(X)$ with small coefficients, we then choose an element $\gamma \in \mathbb{Z}[\theta]$ which has coefficients of order 2^t. This results in a value for d of size approximately $N^N \cdot 2^{t \cdot N}$; thus we require roughly $O(N \cdot (t + \log N))$ bits to represent a single ciphertext.

We first let $T(n)$ denote the function which returns the number of $\mathbb{F}_2$ multiplications needed to perform a multiplication in the field $\mathbb{K}_n = \mathbb{F}_{2^n}$. Using Karatsuba multiplication (for example) we find, for n a power of two, that

$$T(n) := \begin{cases} 1 & \text{if } n = 1, \\ 3 \cdot T(n/2) & \text{otherwise.} \end{cases}$$

This is clearly only an estimate of the overall cost, as we are ignoring the required additions and management of the data.

There are various different options one has for implementing operations on l' finite fields each of size $2^{n'}$. In the following discussion we concentrate on the following four options; clearly other options are available but we select these as a way of demonstrating the different ways how our techniques could be used.

OPTION 1:: We operate on bits using the standard bit-wise FHE schemes, i.e. we take $n = l = 1$ in our FHE scheme. We will then require $l' \cdot n' \cdot t \cdot N$ bits to store our l' finite field elements, and the cost of performing a single SIMD style multiplication on the l' finite fields will cost around $l' \cdot T(n') \cdot T_{\mathsf{bits}}$ multiplications.

OPTION 2:: We operate on the l' finite field elements where each element uses a single ciphertext, i.e. we take $n = n'$ and $l = 1$ in our FHE scheme. This option has the benefit that we can work with the finite field, but we are not forced to operate in a SIMD manner all the time. With such an option we will require $l' \cdot t \cdot N$ bits to store our l' finite field elements, and performing a single SIMD style multiplication on the l' finite fields will cost around $l' \cdot T_{\mathsf{par}}(n', 1)$ multiplications.

OPTION 3:: We operate on all l' finite fields in a SIMD fashion using only a single ciphertext, i.e. we take $n = n'$ and $l = l'$ in our FHE scheme. Thus we will require $t \cdot N$ to store our l' finite field elements, and performing a single SIMD style multiplication on the l' finite fields will cost around $T_{\mathsf{par}}(n', l')$ multiplications.

OPTION 4:: Here we operate on bits, but we operate on them in a SIMD fashion by having a ciphertext represent l' bits, i.e. we take $n = 1$ and $l = l'$ in our FHE scheme. With this option we require $n' \cdot t \cdot N$ bits to store the l' finite field elements, and SIMD style multiplication will require $T(n') \cdot T_{\mathsf{par}}(1, l')$ multiplications.

We summarize the above choices, for the concrete parameters of $n' = 8$ and $l' = 16$, in the following table. We select a value for N around the size of 2000, purely to enable comparison with the work of [12]. We iterate this value is purely for illustrative purposes to show the difference between the various options; it should not be taken to indicate the $N \approx 2000$ is a secure security level. Fixing n', l' and N rather than leaving them variable is done as the overhead of the SIMD operations crucially depends on the specific combination of finite field and cyclotomic field chosen, and has no nice asymptotic meaning. We select a single parameter instance simply not to overwhelm the reader with data, since our goal is purely to show feasibility of our algorithms even at low security levels.

Note, that for Option 1 we select $N = 2048$ since if we are only encrypting bits then using the polynomial $F(X) = X^{2^n} + 1$ will always be more efficient than using $F(X) = \Phi_{3485}(X)$. In addition we keep the parameter t as an indeterminate, as we will be returning to that later.

	N	Ciphertext Space ($\approx$ bits)	Runtime Approx Cost
Option 1	2048	$262144 \cdot t$	$432 \cdot T_{\mathsf{bits}}$
Option 2	2560	$40960 \cdot t$	$16 \cdot T_{\mathsf{par}}(8, 1)$
Option 3	2560	$2560 \cdot t$	$T_{\mathsf{par}}(8, 16)$
Option 4	2560	$20480 \cdot t$	$27 \cdot T_{\mathsf{par}}(1, 16)$

Thus if one is soley interested in reducing the memory of the calculation one would select Option 3. To determine which one is most efficient one needs to actually implement the schemes, since the actual costs of each operation depend on the value of t needed. So we implemented the above algorithms for the four cases $(N, n, l) = (2048, 1, 1)$, $(2560, 8, 1), (2560, 8, 16)$ and $(2560, 1, 16)$, so as to comparre the four options in the above analysis.

In all cases we found that taking $t = 400$ resulted in a scheme in which we were able to recrypt clean ciphertexts; however to enable fully homomorphic encryptions we need to recrypt dirty ciphertexts, and be able to perform some additional operations. For the first two of our four cases we found that $t = 600$ was sufficient, whilst for the second two we found that $t = 800$ was sufficient; note, we increased t in multiples of 100, thus smaller values could have been sufficient.

In the four cases we found the following recrypt times. We also present, assuming we wished in all cases to implement operations on $l' = 16$ values in $\mathbb{F}_{2^{n'}}$, where $n' = 8$, the actual time needed to perform a multiplication in $\mathbb{F}_{2^8}$ followed by a full recrypt, and the total size of all ciphertexts needed to represent such data. In our implementation of the field algorithms for Option 1 and Option 4 we used the Karatsuba method mentioned above, and only performed recryption when implementing a multiplication using the FHE scheme; i.e. recryption was not performed upon additions. The algorithms were implemented in C++ using the NTL library and were run on a machine with six Intel Xeon 2.4 GHz processors and 48 GB of RAM.

Basic FHE Scheme				Performing Ops For $(n', l') = (8, 16)$		
(N, n, l)	t	(p, S)	Recrypt Time (sec)	Method	Mult & Recrypt Time (sec)	Ciphertext Size
$(2048, 1, 1)$	600	$(4, 32)$	15	Option 1	7148	18.00MB
$(2560, 8, 1)$	600	$(4, 32)$	187	Option 2	2983	3.00MB
$(2560, 8, 16)$	800	$(4, 32)$	723	Option 3	735	0.25MB
$(2560, 1, 16)$	800	$(4, 32)$	89	Option 4	2406	2.00MB

The large t value is needed in the last two examples due to the increased complexity of the underlying recryption circuit. We end by noting the following: In our toy example we see that SIMD operations and parallel recryption offer some performance advantages. The exact benefit depends on a number of factors. Firstly the size of n' and l'; these are determined by an application and are often small. In turn n' and l' affect the choice of N, which also depends on the desired security level. The precise values of t and μ allowed are then determined by security analysis of lattice problems. Our toy experiments show that our ability to perform SIMD operations do not affect the size of t very much and that the parallel recryption operation is as practical as standard recryption.

The exact choice of which Option is best however depends on an application. Just as in standard SIMD vs non-SIMD operations on a standard processor, whether one utilizes the SIMD instructions in a program depends on the precise program being run.

8 Possible Applications

Before discussing two possible applications we note that one issue with SIMD operations on data is that sometimes we wish to move data between various elements in the l values on which we are operating. This is often a problem, since the hardware/mathematics/software which supports the SIMD operations precludes such operations. However, in our FHE scheme such operations can be performed at no additional cost.

Indeed given a SIMD word consisting of l elements in a finite field $\mathbb{F}_{2^n}$ one can produce a new SIMD word which consists of any linear function of the bits creating the original SIMD word. To see this we notice that it simply requires multiplying the matrix B used in the parallel recrypt procedure by the matrix defining the linear map. Thus, we can perform this linear function as part of the recryption performed for the previous operation. In particular this means we can shuffle the elements in our SIMD word, or extract specific elements, or extract specific bits, etc. Indeed extracting specific bits in parallel was the core of our parallel recrypt procedure explained above. Note, that this ability to shift around elements and extract elements from a SIMD word is done during the recryption procedure; in the BGV style schemes these operations can be accomplished algebraically on the SIMD word via the use of the homomorphic application of Galois automorphisms, see [13] for further details.

We now turn to our two examples: The first example, namely homomorphic evaluation of AES under some homomorphic key, is used to demonstrate how SIMD operations in high level ($\mathbb{F}_{2^8}$) algebraic structures, allow us to evaluate complex operations relatively easily. Evaluation of AES circuits using FHE operations has been mentioned as a possible usage scenario in [19]. The second example, one of database lookup, provides an example of how data can be searched using SIMD style operations more efficiently than using the bit-wise homomorphic operations envisaged in [11].

In this section we assume that all operations are performed with post-processing by the recryption operation. Thus we are no longer interested in the size of the circuit which implements a functionality but simply the cost of the operations involved. As explained above we have essentially three key operations; the two algebraic operations Mult and Add, plus the linear operations on bits mentioned above. We shall denote the cost of these three operations by C_M, C_A and C_L, and we note that C_L essentially comes for free as part of recryption. For example, if an operation requires two multiplications, one addition and three linear operations we shall denote this cost (for simplicity) by $2 \cdot C_M + C_A + 3 \cdot C_L$.

8.1 Bit-Slicing

Any algorithm which is run on a circuit using bit operations can be run multiple times at once, by executing the algorithm on a set of parameters which supports operations on multiple bits in parallel. Such a technique is often called bit-slicing when applied to a single algorithm; however the technique is essentially also a bit-wise form of SIMD operation. Hence, *any* application performed using an FHE algorithm which supports the parallel recrypt procedure in this paper could be potentially sped-up by at least an order of magnitude by operating on multiple versions of the same algorithm in parallel.

8.2 Application to AES

As an example of the benefits of using SIMD enabled FHE scheme, over the traditional bitwise FHE, we examine the case of how one would implement an AES functionality using FHE. Namely, we want a server to encrypt a message using a key which is only available via an FHE encryption. Using AES as a relatively complex example application of secure computation has also been recently suggested for a number of other related technologies; namely two and multi-party MPC [8, 20]. It is also particularly well suited to SIMD execution due to its overall design. Indeed in [15] the authors extend the ideas of this section to the BGV system; and present actual running times for a fully homomorphic evaluation of the AES circuit.

The method we propose is to encode the entire AES state matrix in a single ciphertext. Recall that the state matrix is a 4-by-4 matrix of elements in $\mathbb{F}_{2^8}$. We therefore first need to select an m so that the ideal (2) splits into at least 16 prime ideals of degree divisible by eight in the field defined by $\Phi_m(X)$. There are a large number of such examples, including the example we have used in this paper of taking $m = 3485$. Note that since $\phi(m)$ is equal to 4×16 we could also perform 4 AES computations in parallel as well, although we will restrict ourselves to one for ease of exposition. In terms of our previous section we let $K_8 = \mathbb{F}_{2^8}$ denote the standard representation of $\mathbb{F}_{2^8}$, i.e.

$$K_8 := \mathbb{F}_2[X]/(X^8 + X^4 + X^3 + X + 1),$$

and we let A denote the algebra consisting of 64 copies of $\mathbb{F}_{2^{40}}$, each with the representation induced by the given factor of $\Phi_m(X) \pmod 2$.

We assume the AES state matrix is given by

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix},$$

which we encode as an element of K_8^{16} as $(s_{0,0}, s_{0,1}, \ldots, s_{3,3})$. Using the map $\Gamma_{8,16}$ we obtain an element of A, which can then be evaluated at α modulo p to obtain a trivial encryption of the message state (before the first round).

To implement AES we assume that the round keys k_i have been presented in encrypted form, using the above embedding via $\Gamma_{8,16}$. Computing the round keys from a given key can be done using the same operations needed to execute the rounds. Thus if we can implement the rounds using efficient Fully Homomorphic SIMD (FH-SIMD) operations, then we can also compute the encryptions of the round keys given the initial key.

The round structure of AES is made up of four basic operations, which we now discuss in turn.

8.2.1 AddRoundKey

This is the simplest operation and is clearly performed for all sixteen bytes in parallel by doing a single $\oplus$ operation of the FHE scheme. This step can be done at the cost of C_A.

8.2.2 ShiftRows

In this operation row i is shifted left by $i - 1$ positions. This is clearly an example of a *linear* operation from earlier, in that we map the ciphertext corresponding to

$$(s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3}, s_{1,0}, s_{1,1}, s_{1,2}, s_{1,3}, s_{2,0}, s_{2,1}, s_{2,2}, s_{2,3}, s_{3,0}, s_{3,1}, s_{3,2}, s_{3,3})$$

into a ciphertext corresponding to

$$(s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3}, s_{1,1}, s_{1,2}, s_{1,3}, s_{1,0}, s_{2,2}, s_{2,3}, s_{2,0}, s_{2,1}, s_{3,3}, s_{3,0}, s_{3,1}, s_{3,2}).$$

Since this is a reordering the cost is given by C_L.

8.2.3 MixColumns

In this step we perform a matrix multiplication on the left of the state matrix by a fixed matrix given by

$$\begin{pmatrix} X & X+1 & 1 & 1 \\ 1 & X & X+1 & 1 \\ 1 & 1 & X & X+1 \\ X+1 & 1 & 1 & X \end{pmatrix}.$$

This is accomplished in four stages

1. Compute the trivial encryption c_1 of $\Gamma_{8,16}((X, X, \ldots, X))$, clearly this can be precomputed.
2. Compute $c_2 \leftarrow c \otimes c_1$.
3. By application of three *linear* operations we can create ciphertexts c_3, c_4, c_5 and c_6 corresponding to c_2 shifted up by one row, c shifted up by one row, c shifted up by two rows, and c shifted up by four rows (where shift rows is performed with rotation).
4. Compute $c_2 \oplus c_3 \oplus c_4 \oplus c_5 \oplus c_6$ and output the result.

Notice that our SIMD operations allows us to perform the 16 multiplications in parallel in the second step. The cost of the MixColumns operation is then $C_M + 4 \cdot C_A + 4 \cdot C_L$.

8.2.4 SubBytes

This is the most complex of all the AES operations, however there is much existing literature on straight line (i.e. no branching) executions of the AES S-Boxes at byte level. For example the approach in [4] transforms the polynomial bases into a "nice" normal basis and then decomposes the arithmetic for inversion into $\mathbb{F}_{2^4}$ and then $\mathbb{F}_{2^2}$ operations. At which point all the arithmetic is just logical operations, and hence amenable to FH-SIMD operations. However, this approach is more suited to real hardware, or to FH-SIMD operations where the basic data type is a bit (e.g. when using say $(n, l) = (1, 16)$ in our main scheme).

As we are restricted to operations which can be performed efficiently in our scheme a more naive approach is probably to be preferred. Recall that the AES S-Box consists of inverting each state byte in K_8 (where we define $0^{-1} = 0$), followed by an $\mathbb{F}_2$-linear operation. Also recall that $x^{-1} = x^{254}$ in the field K_8. We can therefore apply the S-Box operation to our encrypted state using the following method:

- $t \leftarrow c$.
- For $i = 1$ to 6 do
  - $t \leftarrow t \otimes t$.
  - $t \leftarrow t \otimes c$.
- $t \leftarrow t \otimes t$.
- Extract eight ciphertexts $t_0, \ldots, t_7$ such that t_i is the (parallel) encryption of the i-th bit of all 16 values in t.
- Perform the linear operation on $t_0, \ldots, t_7$ in parallel to produce ciphertexts $s_0, \ldots, s_7$.
- Map these ciphertexts back to an encryption of an element in A.

The first step, that of producing an encryption t of x^{254} where c is an encryption of x, requires at most 13 fully homomorphic multiplications. The second step of extracting the ciphertexts $t_0, \ldots, t_7$ is essentially a single linear operation. The third step of adding the elements $t_0, \ldots, t_7$ together to produce $s_0, \ldots, s_7$, requires $4 \cdot 8 = 32$ homomorphic additions, due to the nature of the linear operation in AES. The final step of obtaining a single ciphertext from $s_0, \ldots, s_7$ is also an application of a linear operation. Thus the total cost of SubBytes is given by $13 \cdot C_M + 32 \cdot C_A + 2 \cdot C_L$.

We note that our SIMD evaluation of the AES round function not only benefits in our system from being able to execute 16 operations in parallel. We also have the benefit of being able to deal directly with $\mathbb{F}_{2^8}$ arithmetic operations, as well as decompose into bits where necessary in the linear transformation in the S-Box operation. The total cost of a round function being given by

$$14 \cdot C_M + 37 \cdot C_A + 7 \cdot C_L,$$

although by interleaving operations a lower cost could probably be obtained.

8.3 Data Base Lookup

We end by examining a more realistic application scenario, namely one of searching an encrypted database on a remote server. Suppose a user has previously encrypted a database and stored it on a cloud service provider, and now she wishes to retrieve some of the data. We first note that the usual atomic database operation of search actually consists of two operations. The first operation is one of search, whereas the second is one of retrieval. The following method performs the search using FHE and the retrieval using Private Information Retrieval (PIR).

We assume the database is such that one can determine beforehand which fields will be searched on. In some sense this is akin to the basic premise of public key encryption with keyword search [1], however we have a more complicated data retrieval operation to perform. To simplify the discussion we assume that there is only one database field which is searchable, and another field which contains the information. Each database entry (in the clear) is then given by a tuple (i, s, d), where s is the search term, d is the data and i is some index which is going to enable retrieval. The number of such items we denote by r. We assume that i and s are n bits in length, and thus can be encoded as an element of the finite field $K_n = \mathbb{F}_{2^n}$.

To encrypt the database the user picks a public/private key pair $(\mathsf{pk}, \mathsf{sk})$ for our scheme, as well as a symmetric key K for a symmetric encryption scheme (E_K, D_K). Let us assume that the encryption scheme can support l operations in $\mathbb{F}_{2^n}$ in parallel. When placing the database on the cloud service provider the user divides the database into $\lceil r/l \rceil$ blocks of l items. Then to actually send the server the jth encrypted data block, for $j = 0, 1, 2, \ldots, \lceil r/l \rceil - 1$ we send

$$\begin{aligned}(\mathbf{i}_j, c_j, \mathbf{E}_j) = \big(&i_{l\cdot j+1}, \ldots, i_{l\cdot(j+1)},\\ &\mathsf{Encrypt}(\Gamma_{n,l}(s_{l\cdot j+1}, \ldots, s_{l\cdot(j+1)}), \mathsf{pk}),\\ &E_K(d_{l\cdot j+1}), \ldots, E_K(d_{l\cdot(j+1)})\big)\,.\end{aligned}$$

We now discuss how the user retrieves all data items which correspond to the search term s. We first recover an encryption of an encoding of the index terms which contain this search term. This is done by sending the server one ciphertext, and receiving one in return. The sent "query" ciphertext is equal to

$$q = \mathsf{Encrypt}(\Gamma_{n,l}(s, \ldots, s), \mathsf{pk}),$$

i.e. an encryption of l copies of the query term s.

The server then takes each data block $(\mathbf{i}_j, c_j, \mathbf{E}_j)$ and computes $c_j^{(1)} = q \oplus_{\mathsf{pk}} c_j$. The value $c_j^{(1)}$ is then homomorphically raised to the power $2^n - 1$, by performing $2n$ applications of Mult. This results in a ciphertext $c_j^{(2)}$ which is an encryption of a vector of zero and ones, with a one only occurring in position k when s is not equal to the kth component of the vector underlying the ciphertext c_j.

The server then computes $c_j^{(3)} = (c_j^{(2)} \oplus_{\mathsf{pk}} \mathsf{Encrypt}(\Gamma_{n,l}(1, 1, \ldots, 1), \mathsf{pk})) \otimes_{\mathsf{pk}} \mathsf{Encrypt}(\Gamma_{n,l}(\mathbf{i}_j), \mathsf{pk})$, and the set of ciphertexts $c_j^{(3)}$ are then added together using Add to obtain a final ciphertext c', which is returned to the user. Note, that this "search" query has a cost of $(2 \cdot n + 1) \cdot C_M + 2 \cdot C_A$ per data block.

The plaintext underlying the returned ciphertext c' consists of l components, where the kth component is given by

$$\bigoplus_{s = s_{l\cdot j+k}} i_{l\cdot j+k}.$$

If there is only one match per component then we have recovered the matching indices and hence can recover the actual data by engaging in a PIR protocol [5, 18]. The problem arises when we have the possibility of more than one match per component per query. In this situation we need an encoding algorithm to enable us to recover the exact PIR inputs we need to recover the data.

In the extreme case we have a possibility of every component containing $\lceil r/l \rceil$ matches, i.e. the search term s matches with every item in the database. In which case we obtain, via a trivial encoding, that we must have $\lceil r/l \rceil \leq n$. This essentially implies that the length of the database is bounded by the number of bits we can encrypt, i.e. $r < l \cdot n$.

However, if we can ensure that a maximum of t matches can occur per SIMD component then we can produce a more effective encoding as follows: Firstly we assume the encoding used for data retrieval in the PIR is such that we recover the data item corresponding to an index/component position pair. This simplifies our discussion as we only have to concentrate on decoding a single component.

We set $m = \lceil r/l \rceil$, and to each of the m blocks we associate an n-bit index i. We want to therefore be able, given an xor of the indices $z = i_{j_1} \oplus \ldots \oplus i_{j_s}$, with $s \leq t$, to recover the set $\{i_{j_1}, \ldots, i_{j_s}\}$. To construct the encoding we take the parity matrix of an $[N, K, D]$ linear code over $\mathbb{F}_2$ of length N, rank K and minimum distance D, which we assume is greater $2 \cdot t$. This is a matrix of dimension $(N - K) \times N$. We then take as our indices the columns of this matrix, which implies that these indices must fit in n bits, hence $N - K \leq n$. Given an xor of at most t indices we can recover

which indices were xor-ed together by decoding the $[N, K, D]$ linear code. To see this notice that the sum of indices z is a syndrome of a codeword in the linear code. Thus by recovering the error positions in the code from the syndrome we know which indices, i.e. which columns of the parity check matrix, were xor-ed together. Thus the total number of distinct indices we can cope with is bounded by the column size of the parity check matrix, i.e. N. Hence, we obtain $m = \lceil r/l \rceil \leq N$.

As an example of a possible encoding scheme we take a primitive BCH code which exists for any pair of values of (s, t) such that $s \geq 3$ and $t < 2^{s-1}$. The primitive BCH code over $\mathbb{F}_2$ then has parameters given by $N = 2^s - 1$, $N - K \leq s \cdot t$ and $D \geq 2 \cdot t + 1$. If we take our FHE scheme of the previous section using the mth cyclotomic polynomial with $m = 3485$, then we have $l = 64$, $n \leq d = 40$ and $\phi(m) = 2560$. Given the bounds

$$\lceil r/l \rceil \leq N = 2^s - 1 \text{ and } s \cdot t \leq n,$$

and supposing we take $t = 3$, so we can recover at most three collisions on search terms within each component, then by setting $n = d = 40$ and $(s, t) = (13, 3)$ we obtain a valid encoding. This implies that the total number of items within the database is bounded by $l \cdot N = 524224$. Clearly using more optimal codes, or different cyclotomic polynomials one can obtain larger values of the whole database, or one can deal with more collisions within a component.

The above methodology using our SIMD enabled FHE scheme to search on l components at once in an efficient manner, results in a linear speed up in the search of the encrypted database. However, there is another advantage of our splitting the database into l components; we can deal with (albeit having a probability of invalid indices being returned) having more collisions between the search terms. In the above example we could deal with up to three collisions in each component, this meant that our method would be guaranteed to be correct if there were at most three items in the database corresponding to each search item. However, if we assume that the search items are randomly distributed between the l components, then in practice we can deal with more collisions, since our results will be correct as long as there are at most t collisions *per component*. The generalised birthday bound [24] says that we can have

$$(t!)^{1/t} \cdot l^{(t-1)/t}$$

collisions before the probability of obtaining more than t collisions in one of the l components is greater than $1/2$. In our above numerical example, with $t = 3$ and $l = 64$, this equates to just over 29 matches in our database.

9 Acknowledgements

This material is based on research sponsored by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II. The first author was also supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079, by the Royal Society via a Royal Society Wolfson Merit Award, by the ERC via Advanced Grant ERC-2010-AdG-267188-CRIPTO, and the EPSRC via grant EP/I03126X. The second author was supported by a Postdoctoral Fellowship of the Research Foundation - Flanders (FWO).

The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

1. D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. *Advances in Cryptology – Eurocrypt 2004*, Lecture Notes in Comput. Sci. **3027**, 506–522, 2004.
2. Z. Brakerski, C. Gentry and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Innovations in Theoretical Computer Science –ITCS 2012*, 309–325, ACM, 2012.
3. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from Ring-LWE and security for key dependent messages. *Advances in Cryptology – Crypto 2011*, Lecture Notes in Comput. Sci. **6841**, 505–524, 2011.
4. D. Canright. A very compact S-Box for AES. *Cryptographic Hardware and Embedded Systems – CHES 2005*, Lecture Notes in Comput. Sci. **3659**, 441–455, 2005.
5. B. Chor, E. Kushilevitz, O. Goldreich and M. Sudan. Private information retrieval. *J. ACM*, **45**, 965–981, 1998.
6. M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. *Advances in Cryptology – Eurocrypt 2010*, Lecture Notes in Comput. Sci. **6110**, 24–43, 2010.
7. J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, **19**, 297–301, 1965.
8. I. Damgård and M. Keller. Secure multiparty AES. *Financial Cryptography – FC 2010*, Lecture Notes in Comput. Sci. **6052**, 367–374, 2010.
9. I. Damgård, V. Pastro, N.P. Smart and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. To appear *Advances in Cryptology – Crypto 2012*.

10. C. Gentry. Fully homomorphic encryption using ideal lattices. *Symposium on Theory of Computing – STOC 2009*, ACM, 169–178, 2009.
11. C. Gentry. A fully homomorphic encryption scheme. *Manuscript*, 2009.
12. C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. *Advances in Cryptology – Eurocrypt 2011*, Lecture Notes in Comput. Sci. **6632**, 129–148, 2011.
13. C. Gentry, S. Halevi and N.P. Smart. Fully homomorphic encryption with polylog overhead. *Advances in Cryptology – Eurocrypt 2012*, Lecture Notes in Comput. Sci. **7237**, 465–482, 2012.
14. C. Gentry, S. Halevi and N.P. Smart. Better bootstrapping in fully homomorphic encryption. *Public Key Cryptography – PKC 2012*, Lecture Notes in Comput. Sci. **7293**, 1–16, 2012.
15. C. Gentry, S. Halevi and N.P. Smart. Homomorphic evaluation of the AES circuit. To appear *Advances in Cryptology – Crypto 2012*.
16. C. Gentry, S. Halevi and N.P. Smart. Ring switching in BGV-style homomorphic encryption. IACR ePrint 2012/240, `http://eprint.iacr.org/2012/240/`.
17. I.J. Good. The interaction algorithm and practical Fourier analysis. *J.R. Stat. Soc.*, **20**, 361–372, 1958.
18. E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. *Foundations of Computer Science – FoCS '97*, 364–373, 1997.
19. K. Lauter, M. Naehrig, V. Vaikuntanathan. Can homomorphic encryption be practical? *Cloud Computing Security Workshop – CCSW 2011*, 113–124, ACM, 2011.
20. B. Pinkas, T. Schneider, N.P. Smart, S.C. Williams. Secure two-party computation is practical. *Advances in Cryptology – Asiacrypt 2009*, Lecture Notes in Comput. Sci. **5912**, 250-267, 2009.
21. C.M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proc. IEEE*, **56**, 1107–1108, 1968.
22. P. Scholl and N.P. Smart. Improved key generation for Gentry's fully homomorphic encryption scheme. *Cryptography and Coding – IMACC 2011*, Lecture Notes in Comput. Sci. **7089**, 10–22, 2011.
23. N.P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. *Public Key Cryptography – PKC 2010*, Lecture Notes in Comput. Sci. **6056**, 420–443, 2010.
24. K. Suzuki, D. Tonien, K. Kurosawa and K. Toyota. Birthday paradox for multi-collisions. *Information Security and Cryptology – ICISC 2006*, Lecture Notes in Comput. Sci. **4296**, 29–40, 2006.
25. L.H. Thomas. Using a computer to solve problems in physics. *Application of Digital Computers*, 1963.

Improved Key Generation For Gentry's Fully Homomorphic Encryption Scheme

P. Scholl and N.P. Smart

Dept. Computer Science,
University of Bristol,
Woodland Road,
Bristol, BS8 1UB,
United Kingdom.

Abstract. A key problem with the original implementation of the Gentry Fully Homomorphic Encryption scheme was the slow key generation process. Gentry and Halevi provided a fast technique for 2-power cyclotomic fields. We present an extension of the Gentry–Halevi key generation technique for arbitrary cyclotomic fields. Our new method is roughly twice as efficient as the previous best methods. Our estimates are backed up with experimental data.

The major theoretical cryptographic advance in the last three years was the discovery by Gentry in 2009 of a fully homomorphic encryption scheme [4, 5]. Gentry's scheme was initially presented as a completely theoretical construction, however it was soon realised that by specialising the construction one could actually obtain a system which could at least be implemented; although not yet in such a way as to enable practical computations. The first such implementation was presented by Smart and Vercauteren [10]. The Smart and Vercauteren implementation used arithmetic of cyclotomic number fields. In particular they focused on the field generated by the polynomial $F(X) = X^{2^n} + 1$, but they noted that the scheme could be applied with arbitrary (even non-cyclotomic) number fields. A main problem with the version of Smart and Vercauteren was that the key generation method was very slow indeed.

In [6] Gentry and Halevi presented a new implementation of the variant of Smart and Vercauteren, but with a greatly improved key generation phase. In particular Gentry and Halevi note that key generation (for cyclotomic fields) is essentially an application of a Discrete Fourier Transform, followed by a small amount of computation, and then application of the inverse Discrete Fourier Transform. They then show that one does not even need to perform the DFT's if one selects the cyclotomic field to be of the form $X^{2^n} + 1$. They do this by providing a recursive method to deduce two constants, from the secret key, which enables the key generation algorithm to construct a valid associate public key. The key generation method of Gentry and Halevi is fast, but appears particularly tailored to working with two-power roots of unity.

However, the extra speed of their key generation method comes at a cost. Restricting to two-power roots of unity means that one is precluded from the

type of SIMD operations discussed in [11]. To enable such operations one needs to be able to deal with general cyclotomic number fields. In [11] it is pointed out that the DFT/inverse-DFT method can be easily applied to the case of general cyclotomic fields via the use of the FFT algorithms such as those of Good–Thomas [7, 12], Rader [9] and others. However, the simple recursive method of Gentry and Halevi does not seem to apply.

Other works have examined ways of improving key generation, and fully homomorphic encryption schemes in particular. For example [8] has a method to construct keys for essentially random number fields by pulling random elements and analyzing eigenvalues of the corresponding matrices; this method however does not allow the efficiency improvements of [10] and [6] with respect to reduced ciphertext sizes etc. More recent fully homomorphic schemes based on the LWE assumption [3] have more efficient key generation procedures than the original Gentry scheme; and appear to be more suitable in practice. However for this work we concentrate purely on the schemes in the "Gentry family".

In this paper we present an analysis of the key generation algorithm, for Gentry based schemes, for general cyclotomic fields, generated by the the primitive m-th roots of unity. In particular, we show that Gentry and Halevi's recursive method can be generalised to deal with prime power values of m, and also any m with just a few small, repeated prime factors. We also show for general m that the DFT/inverse-DFT method is sub-optimal, and that an algorithm exists which requires only a single DFT application to compute the secret key. Our general key generation method is essentially twice as fast as previous methods; both theoretically and in practice.

The paper is organized as follows: In Section 1 we present the required mathematical background and notation. In Section 2 we present the required information about the key generation method for the variant of Gentry's scheme we will be discussing. Then in Section 3 we describe how one could execute the key generation procedure assuming as soon as two coefficients of one associated polynomial $g(X)$ and one coefficient of another associated polynomial $h(X)$ are computed. Algorithms to compute these three coefficients are then presented in Section 4. Finally in Section 5 we present some experimental results.

1 Mathematical Background

Let $F(X) = \Phi_m(X)$ denote the m-th cyclotomic polynomial, i.e. the irreducible polynomial whose roots are the primitive m-th roots of unity. This polynomial has degree $N = \phi(m)$, where $\phi(\cdot)$ is Euler's phi-function. We let the m-th roots of unity be denoted by $\omega_m^0, \ldots, \omega_m^{m-1}$, which are defined as powers of $\omega_m = \exp(\frac{2\pi\sqrt{-1}}{m})$, the principal m-th root of unity. The roots of $F(X)$ are those values ω_m^i where $\gcd(i, m) = 1$. We let $\rho_0, \ldots, \rho_{N-1}$ denote these primitive m-th roots of unity (i.e. the roots of F).

If $f(X) \in \mathbb{Z}[X]$ is an arbitrary polynomial then we let f_i denote the coefficient of X^i in $f(X)$. For a polynomial $f(X)$ we let $\|f\|_\infty = \max_{i=0}^{\deg(f)} |f_i|$ denote the infinity-norm (i.e. the max-norm) of its coefficient vector. Given two polynomials

$f(X)$ and $g(X)$ the resultant of f and g is defined to be

$$\mathsf{resultant}(f,g) = \prod_{\alpha,\beta}(\alpha - \beta)$$

where α ranges over the roots of $f(X)$ and β ranges over the roots of $g(X)$. We also have that

$$\mathsf{resultant}(f,g) = \prod_{\alpha} g(\alpha). \tag{1}$$

Given a polynomial $x(X)$ of degree $m-1$, which is simply a list of coefficients $x_0, x_1, \ldots, x_{m-1}$, the Discrete Fourier Transform (DFT) is defined by the evaluation of this polynomial at all of the m-th roots of unity. So the k-th coefficient of the DFT is then

$$\mathbf{x}_k = \sum_{i=0}^{m-1} x_i \omega_m^{i \cdot k}.$$

Naïve computation of the DFT from this definition takes $O(m^2)$ operations. Fast Fourier Transform (FFT) algorithms reduce this to $O(m \log m)$. The inverse-DFT is the procedure which takes m evaluations of a polynomial at the m-th roots of unity, and then recovers the polynomial. We write $\mathbf{x} \leftarrow \mathsf{DFT}(x)$ and $x \leftarrow \mathsf{DFT}^{-1}(\mathbf{x})$.

2 Key Generation for Gentry

Key generation for Gentry's FHE scheme depends on two parameters m and t. The value m defines the underlying cyclotomic field as above, and we define $N = \phi(m)$, which is the degree of the cyclotomic polynomial $F(X)$. The parameter t is used to define how "small" the secret key is. Note that in practice the word "small" is a relative term and we are not really dealing with small numbers at all. To generate keys for Gentry's FHE scheme one can proceed as follows:

- $v(X) \leftarrow \mathbb{Z}[X]$ with $\|v\|_\infty \leq 2^t$ and $v(X) \equiv 1 \pmod 2$.
- Compute $w(X) \in \mathbb{Z}[X]$ such that

$$d = v(X) \cdot w(X) \pmod{F(X)}$$

  where $d = \mathsf{resultant}(v, f)$.
- If $v(X)$ and $w(X)$ do not have a common root modulo d then return to the beginning and choose another $v(X)$.
- Let $\alpha \in \mathbb{Z}_d$ denote the common root.
- Set $\mathfrak{pk} \leftarrow (\alpha, d)$ and $\mathfrak{sk} \leftarrow (w(X), d)$.

Note, there are various minor variations on the above procedure in the literature. In Smart and Vercauteren [10] the polynomial $v(X)$ is rejected unless d is prime; this is done due to the method the authors used to compute the common root α. Gentry and Halevi [6] notice that if $v(X)$ and $f(X)$ have a common root modulo $f(X)$ then it is given by $\alpha = -w_{N-1}/w_0 \pmod d$. Gentry and Halevi, make an

additional modification, in that the condition on $v(X) \equiv 1 \pmod 2$ is dropped, and replaced by the condition that $d \equiv 1 \pmod 2$; this means the authors only need to compute one coefficient of $w(X)$ for their application. However, in [11], the authors show that selecting $v(X) \equiv 1 \pmod 2$ enables SIMD style operations on data, as long as $m \neq 2^r$. They also show that whilst all coefficients of $w(X)$ are needed in the secret key, one can generate all of them via the relation

$$w_i = \begin{cases} \alpha w_{i+1} + F_{i+1} w_{N-1} & \pmod d \quad \text{if } 0 \leq i < N-1 \\ -\alpha w_0 & \pmod d \quad \text{if } i = N-1 \end{cases} \tag{2}$$

The main question is then how to compute w_0 and d. In [6, 11] it is pointed out that the following DFT-based procedure can be applied:

- $\mathbf{v} \leftarrow \mathsf{DFT}(v(X))$.
- $d \leftarrow \prod_{\gcd(i,m)=1} \mathbf{v}_i$.
- $\mathbf{w}_i \leftarrow d/\mathbf{v}_i$.
- $w(X) \leftarrow \mathsf{DFT}^{-1}(\mathbf{w})$.

Gentry and Halevi [6] then go on to notice that one can actually compute $w(X)$ and d without any need for computing DFTs. They do this, since they solely focus on the case $m = 2^r$, which enables them to present the calculation of d and $w(X)$ as the calculation of computing two coefficients of an associated polynomial $g(X)$.

In this paper we generalise this method of Gentry and Halevi to arbitrary values of m; for non-prime powers of m we will still require the application of a single DFT algorithm, but will no longer need the inverse DFT. The key observation is that d and $w(X)$ are related, for general m, to the coefficients of *two* associated polynomials $g(X)$ and $h(X)$. It is to these polynomials, and their properties, that we now turn.

3 The Polynomials $g(X)$ and $h(X)$

Before proceeding we introduce Ramanujan sums, for those readers who are not acquainted with them. A Ramanujan sum is simply a sum of powers of primitive roots of unity:

$$C_m(k) := \sum_{\substack{i=0 \\ (i,m)=1}}^{m-1} \omega_i^k = \sum_{d|(k,m)} \mu\left(\frac{m}{d}\right) d$$

where the second sum is over the positive divisors of $\gcd(k, m)$, and μ is the Möbius function. For a proof of this formula see e.g. [2, p. 162]. The Ramanujan sum can therefore be easily computed provided m can be factored efficiently; this will always be the case in our applications since m is a small integer. It is clear from this formula that $C_m(-k) = C_m(k)$. We also have the following result, which we will need:

Proposition 1. *Let F_i denote the i-th coefficient of the m-th cyclotomic polynomial $F(X)$. Then for $k = 0, \ldots, N-1$,*

$$\sum_{i=1}^{N-1} C_m(i-k) \cdot F_{i+1} = -C_m(-k-1).$$

Proof. Suppose that θ is a root of F. Observing that since F is a cyclotomic polynomial, $F_0 = F_N = 1$, and so

$$-1 = \sum_{i=1}^{N} F_i \theta^i = \sum_{i=0}^{N-1} F_{i+1} \theta^{i+1}.$$

This is equivalent to

$$-\theta^{-k-1} = \sum_{i=0}^{N-1} F_{i+1} \theta^{i-k}.$$

The above relation can then be applied to the individual summands in $C_m(k)$ (which are powers of the roots of F) to give the desired result.

We now turn to our key generation method. Given $v(X)$ we define the following polynomials,

$$g(X) := \prod_{i=0}^{N-1} (v(\rho_i) - X)$$
$$h(X) := \prod_{i=0}^{N-1} (v(\rho_i) - X/\rho_i).$$

The polynomial g here is the same as that defined in [6]. However, when m is not a power of 2 we also need to introduce $h(X)$ in order to help us find w.

The constant-term and degree one coefficients of these polynomials, i.e. g_0, g_1, h_0 and h_1, must then be computed. We leave discussion of how this step is done until the next section. In this section we detail how, given these coefficients, we can compute $w(X)$ and d. Note that because of Equation 1, the values g_0 and h_0 are both equal to the resultant, d, of v and f.

We also have

$$g_1 = -\sum_{i=0}^{N-1} \prod_{j \neq i} v(\rho_j) = -\sum_{i=0}^{N-1} \frac{\prod_{j=0}^{N-1} v(\rho_j)}{v(\rho_i)} = -\sum_{i=0}^{N-1} \frac{d}{v(\rho_i)} = -\sum_{i=0}^{N-1} w(\rho_i) \quad (3)$$

and similarly,

$$h_1 = -\sum_{i=0}^{N-1} \frac{w(\rho_i)}{\rho_i}. \quad (4)$$

To determine the coefficients of w, we first look at a more general form of the above expressions for g_1 and h_1, and show how this relates to w. Define for $k \geq 0$ the following sequence of sums

$$W_k := \sum_{i=0}^{N-1} \frac{w(\rho_i)}{\rho_i^k}.$$

Our strategy from here onwards is to give a simple expression for W_k in terms of the coefficients of w, and then show that the values of W_k can be easily computed independently using the information we already have of g_1 and h_1. Next, by looking at successive terms of W_k, a set of simultaneous equations involving the coefficients of w will arise, and it will be shown that these can be solved to recover all of w.

Observe that, as a result of Equations 3 and 4, we have $W_0 = -g_1, W_1 = -h_1$. More generally, we see that

$$W_k = \sum_{i=0}^{N-1} \frac{\sum_{j=0}^{N-1} w_j \cdot \rho_i^j}{\rho_i^k} = \sum_{j=0}^{N-1} w_j \cdot \sum_{i=0}^{N-1} \rho_i^{j-k} = \sum_{j=0}^{N-1} C_m(j-k) \cdot w_j.$$

Thus the above equation gives us an expression for W_k as a simple linear combination of the coefficients of w, by the Ramunujan sums $C_m(j-k)$. Applying Equation 2, this allows us to deduce

Proposition 2.

$$W_k = \alpha \cdot W_{k+1} \pmod{d}.$$

Proof.

$$\begin{aligned}
W_k &= \sum_{i=0}^{N-1} C_m(i-k) \cdot w_i \\
&= \sum_{i=0}^{N-2} C_m(i-k) \cdot \alpha \cdot w_{i+1} + w_{N-1} \cdot \sum_{i=0}^{N-2} C_m(i-k) \cdot F_{i+1} \\
&\quad + C_m(N-k-1) \cdot w_{N-1} \\
&= \alpha \cdot \sum_{i=0}^{N-2} C_m(i-k) \cdot w_{i+1} + w_{N-1} \cdot \sum_{i=0}^{N-1} C_m(i-k) \cdot F_{i+1} \\
&= \alpha \cdot \sum_{i=1}^{N-1} C_m(i-k-1) \cdot w_i - w_{N-1} \cdot C_m(-k-1) \\
&= \alpha \cdot \sum_{i=1}^{N-1} C_m(i-k-1) \cdot w_i + \alpha \cdot w_0 \cdot C_m(-k-1) \\
&= \alpha \cdot W_{k+1}
\end{aligned}$$

From which comes the following immediate corollary:

Corollary 1.

$$W_k = -g_1 \cdot \alpha^{-k} \pmod{d}.$$

Note that Proposition 2 immediately implies that $\alpha = g_1/h_1 \bmod d$, and thus any value of W_k can be easily determined using the corollary. This allows us to create a system of linear equations in the coefficients of w, from the values of $W_0, \ldots, W_{N-1}$, as follows:

$$\begin{pmatrix} C_m(0) & C_m(1) & \cdots & C_m(N-1) \\ C_m(1) & C_m(0) & \cdots & C_m(N-2) \\ \vdots & \vdots & \ddots & \vdots \\ C_m(N-2) & C_m(N-3) & \cdots & C_m(1) \\ C_m(N-1) & C_m(N-2) & \cdots & C_m(0) \end{pmatrix} \cdot \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N-1} \end{pmatrix}$$

$$= -g_1 \cdot \begin{pmatrix} 1 \\ \alpha^{-1} \\ \alpha^{-2} \\ \vdots \\ \alpha^{1-N} \end{pmatrix} \pmod{d}$$

We write the above equation as $C \cdot \boldsymbol{w} = -g_1 \cdot \boldsymbol{\alpha}$. The matrix C possesses the interesting property that every diagonal is constant; as such it is a symmetric Toeplitz matrix. There is a method to solve such a system of equations in only $O(N^2)$ operations, as opposed to the usual $O(N^3)$ required for a general matrix [13]. We note, that for a given value of m the matrix C is fixed and hence computing its inverse can be considered as a precomputation. Thus with this precomputation the cost of computing the key, given the coefficients g_0, g_1, and h_0, is a linear operation in N.

When it comes to computing the inverse of the matrix C, we note that it appears experimentally to be of the form, for all m,

$$C^{-1} = \frac{1}{m} Z,$$

for some integral $N \times N$ matrix Z whose coefficients are bounded in absolute value by m. However, we were unable to prove this. In any case we can assume this is true, then efficiently compute the inverse of C by inverting C/m using standard floating point arithmetic and then rounding the resulting coefficients to integers. This matrix can then be divided by m, tested for correctness and stored.

4 Determining g_0, g_1 and h_1

In this section we examine methods to determine the coefficients g_0, g_1 and h_1. We first present a general method, which works for arbitrary values of m and

leads to key generation that is essentially twice as fast as existing methods. We then describe a method for "special" values of m, namely those containing a large number of repeated factors, such as when m is a prime power. By specialising the results of this section, and the method in the previous section to the case $m = 2^r$, we obtain the key generation method of Gentry and Halevi.

4.1 General m

We note that the desired coefficients of g and h can be computed directly from the FFT of v. Thus by applying one FFT and the techniques of the previous section we can avoid the second inverse-FFT required of the method in Section 2. Hence, we can obtain a method which is essentially twice as fast as that proposed in 2.

Recall that the FFT of v gives the values $v(\rho_0), v(\rho_1), \ldots, v(\rho_{N-1})$. With these computed, g_0 is obtained by simply multiplying them together (as is done in the FFT-based key generation algorithm). Then note that

$$g_1 = -\sum_{i=0}^{N-1} \frac{g_0}{v(\rho_i)}$$

and

$$h_1 = -\sum_{i=0}^{N-1} \frac{g_0}{\rho_i \cdot v(\rho_i)}.$$

So the coefficients g_1 and h_1 can all be computed in $O(N)$ operations (albeit on numbers of $O(N \cdot t)$ bits in length), once the initial FFT of v is computed. This may not seem a major improvement, after all we have only really saved one FFT out of two; but there is a huge implied constant in the big-Oh notation due to the fact that the coefficients of the polynomial $w(X)$ are all of size around $2^{N \cdot t}$, which is practice will result in many millions of bits of precision being needed in the FFT algorithms.

4.2 The case $m = p^r$

We first define the following two polynomials

$$a(X) = \prod_{j=0}^{p-1} v(\alpha_j \cdot X)$$

$$b(X) = \sum_{j=0}^{p-1} \prod_{j \neq i} v(\alpha_j \cdot X).$$

where $\alpha_0, \ldots, \alpha_{p-1}$ denote the p-th roots of unity. By elementary Galois theory we find that the coefficients of a must be rational integers. We observe that $a(\alpha_i \cdot X) = a(X)$, so it must follow that the i-th coefficient of a will be zero if i

is not a multiple of p. By a similar argument we also deduce that $b(X) \in \mathbb{Z}[X]$ and that $b_i = 0$ if i is not a multiple of p.

Our algorithm will depend on starting with the polynomials $a(X)$ and $b(X)$. These can be easily computed due to the following observations. Firstly, by [1, Proposition 4.3.4], we have

$$a(X^p) = p^{1-p} \cdot \mathsf{resultant}_Y(v(Y), p \cdot X - p \cdot Y^p).$$

where $\mathsf{resultant}_Y(f, g)$ denotes the resultant polynomial in Y of the bivariate polynomials f and g. Note that when computing this resultant, every occurrence of Y^p in the polynomial $v(Y)$ can be replaced with X to vastly speed up computation time.

Now notice also that

$$b(X) = \sum_{i=0}^{p-1} \frac{a(X)}{v(\alpha_i \cdot X)} = \sum_{i=0}^{p-1} \frac{a(\alpha_i \cdot X)}{v(\alpha_i \cdot X)} = \sum_{i=0}^{p-1} (a/v) \cdot (\alpha_i X).$$

Then by writing $(a/v)(X) = \sum_{j=0}^{N-1} B_j \cdot X^j$ and changing the order of summations, we obtain:

$$b(X) = \sum_{j=0}^{N-1} B_j \cdot X^j \cdot \left(\sum_{i=0}^{p-1} \alpha_i^j \right) = p \sum_{j=0}^{N/p-1} B_{p \cdot j}.$$

So the polynomial $b(x)$ can be computed from the coefficients of the quotient polynomial a/v; note that this is an exact polynomial division over $\mathbb{Z}[X]$.

Now recall the definition of g, in terms of v evaluated at the primitive roots of unity:

$$g(X) := \prod_{i=0}^{N-1} (v(\rho_i) - X).$$

Since $m = p^r$, it can be shown that the primitive m-th roots of unity are heavily related to the p-th roots of unity, $\alpha_0, \ldots, \alpha_{p-1}$. For any $k \in \{0, \ldots, p-1\}$,

$$\rho_{i+k \cdot N/p} = \alpha_k \cdot \rho_i.$$

Using this fact, the length-$(N-1)$ product defining g above can be re-expressed as a length-$(N/p-1)$ product of p-products, involving the p-th roots of unity. Applying this to g and then evaluating modulo X^2 (to obtain the lowest two

coefficients) gives

$$\begin{aligned} g(X) &= \prod_{i=0}^{N/p-1} \prod_{j=0}^{p-1} (v(\alpha_j \cdot \rho_i) - X) \\ &= \prod_{i=0}^{N/p-1} \left(\underbrace{\prod_{j=0}^{p-1} v(\alpha_j \cdot \rho_i)}_{a(\rho_i)} - X \cdot \underbrace{\sum_{j=0}^{p-1} \prod_{j \neq i} v(\alpha_j \cdot \rho_i)}_{b(\rho_i)} \right) \pmod{X^2} \\ &= \prod_{i=0}^{N/p-1} \big(a(\rho_i) - X \cdot b(\rho_i)\big) \pmod{X^2}. \end{aligned}$$

Since $a(X)$ and $b(X)$ are integer polynomials whose i-th coefficient is zero if p does not divide i, and that $F(X)$ (the p^r-th cyclotomic polynomial) has non-zero coefficients only for coefficients of X to the power of some multiple of p^{r-1}, we have that $a'(X) := a(X) \pmod{F(X)}$ and $b'(X) := b(X) \pmod{F(X)}$ will also be polynomials whose i-th coefficient is zero if p does not divide i.

So, if we define the polynomials V, U, such that $V(X^p) = a(X) \pmod{F(X)}$ and $U(X^p) = b(X) \pmod{f(X)}$, then we have reduced the original product of length N over v of degree $N-1$ down to a product of length N/p over the polynomials V and U, which have degree $N/p - 1$. This process can be applied recursively, until we end up with a final product of size $N/p^{r-1} = p-1$. This last product can then be computed in the naïve manner to obtain $g(X) \pmod{X^2}$. A similar reduction can also be applied to h.

The algorithm in Figure 1 shows how this reduction can be applied to compute g_0 and g_1. A simple modification to the algorithm will also allow h_1 to be computed at the same time. The proof of correctness for this is an obvious generalisation of the proof for the Gentry and Halevi reduction [6] and so is omitted.

4.3 m contains repeated factors

The algorithm described above can be used to speed up computation of g and h whenever m contains a repeated prime factor. If $m = p_1^{r_1} \cdots p_s^{r_s}$, then for every $r_i > 1$, $r_i - 1$ steps of the algorithm in Figure 1 can be carried out. So after each of these reductions the final product to be computed will be of size $(p_1 - 1) \cdots (p_s - 1)$. Clearly this speed improvement is most pronounced when $m = p^r$ for some small p, but it is nevertheless useful to note that gains can be made for any m with repeated prime factors.

5 Experiment Results

We now present some computational results for the relative performance of our new key generation method compared to the previous version. The original method was implemented in C++ using the MPFR library for arbitrary

COMPUTE-g-COEFFICIENTS(v, p, r)

$m \leftarrow p^r$
$F(X) \leftarrow \Phi_m(X)$
$U(X) \leftarrow 1$
$V(X) \leftarrow v(x)$
while $m > p$
    $v(X) \leftarrow V(X) \pmod{F(X)}$
    $V(X) \leftarrow \mathsf{resultant}_Y(v(Y),\, p \cdot X - p \cdot Y^p)/p^{p-1}$
    $q(X) \leftarrow U(X) \cdot V(X^p)/v(X)$
    for $i \leftarrow 0$ **to** $\deg(q)/p$
        $U_i \leftarrow q_{p \cdot i}$
    $U(X) \leftarrow U(X) \pmod{F(X)}$
    $U(X) \leftarrow U(X^{1/p})$
    $m \leftarrow m/p$
    $F(X) \leftarrow \Phi_m(X)$
// After the reduction, $p-1$ terms are left in the product.
$\rho \leftarrow e^{2 \cdot \pi \cdot \sqrt{-1}/p}$.
$g_0 \leftarrow \prod_{i=1}^{p-1} V(\rho^i), \qquad g_1 \leftarrow \sum_{i=1}^{p-1} U(\rho^i) \prod_{j \neq i} V(\rho^i)$
return g_0, g_1

Fig. 1. Algorithm to compute g_0 and g_1 when $m = p^r$.

precision floating point arithmetic, compiled using GCC 4.3.5. Our new method was coded with the computer algebra system Sage. Both algorithms were run on a high-powered server featuring an Intel Xeon E5620 processor running at 2.4GHz, with a 12MB cache and 48GB of memory.

We first describe the performance at four different values of m, each with different factorization properties. Namely, $m = 4391$, 5555, 6561 and 10125, which result in values of $n = \phi(m)$ in the range $[4000, 5400]$. The results (in minutes) for a value of $t = 400$ are given in Table 1.

m	4391	5555 $(= 5 \cdot 11 \cdot 101)$	6561 $(= 3^8)$	10125 $(= 3^4 \cdot 5^3)$
$\phi(m)$	4390	4000	4374	5400
Original Method	274	137	204	451
New Method	164	67	30	123
% Improvement	40%	51%	85%	72%

Table 1. Comparison of key generation methods for $t = 400$ and various values of m. Times are in minutes.

In Figure 2, we show how the performance of each algorithm as affected by t, for a fixed choice of m. We test each m with several different choices of the parameter t, the bit size of the generated coefficients. The bit length of a key will be approximately $t \cdot \phi(m)$, so increasing t increases the size of the numbers being computed on, and also requires a greater precision for any necessary floating point operations.

It is clear that our new method is significantly faster than the FFT method for all choices of m. In particular, when m contains many small repeated factors (here, for $m = 6561$ and 10125) the improvement gained is almost an order of magnitude. When the hybrid approach is taken, we see that the cost of recovering the key by inverting the matrix is far lower than that of using the second (inverse) FFT in the standard FFT method, and results in a speed increase of around 40-50%, as expected.

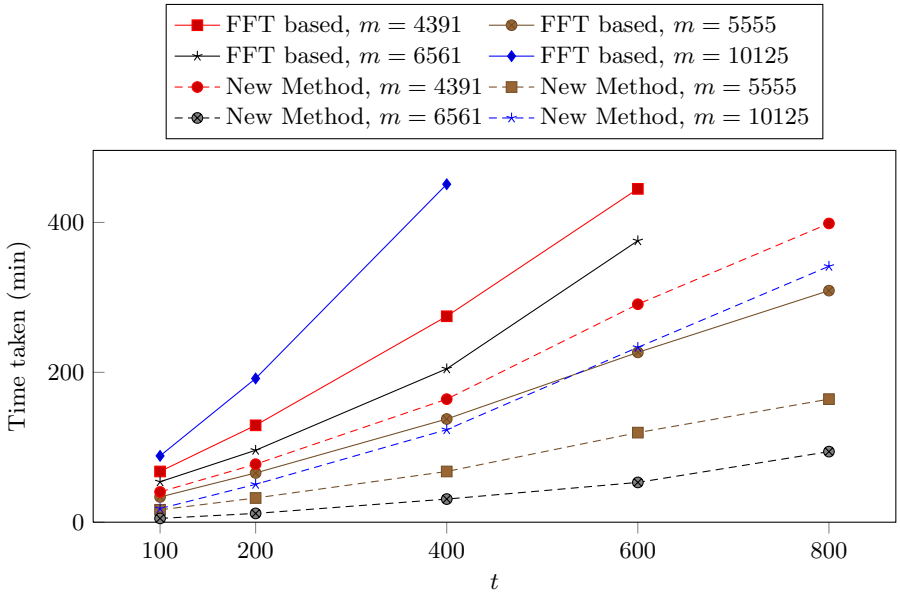

Fig. 2. Comparison of methods for various different values of m, as the parameter t increases.

6 Acknowledgements

The second author was supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant COED–EP/I03126X, the Defense Advanced Research Projects Agency (DARPA) and

the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079, and by a Royal Society Wolfson Merit Award. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, AFRL, the U.S. Government, the European Commission or EPSRC.

References

1. H. Cohen. A Course in Computational Algebraic Number Theory. Springer GTM 138, 1993.
2. T.M. Apostol Introduction to Analytic Number Theory. Springer-Verlag, New York, 1976.
3. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. *Advances in Cryptology – Crypto 2011*, Springer LNCS 6841, 505–524, 2011.
4. C. Gentry. Fully homomorphic encryption using ideal lattices. *Symposium on Theory of Computing – STOC 2009*, ACM, 169–178, 2009.
5. C. Gentry. A fully homomorphic encryption scheme. PhD, Stanford University, 2009.
6. C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. *Advances in Cryptology – Eurocrypt 2011*, Springer LNCS 6632, 129–148, 2011.
7. I.J. Good. The interaction algorithm and practical Fourier analysis. *J.R. Stat. Soc.*, **20**, 361–372, 1958.
8. N. Ogura, G. Yamamoto, T. Kobayashi and S. Uchiyama. An improvement of key generation algorithm for Gentry's homomorphic encryption scheme. *Advances in Information and Computer Security – IWSEC 2010*, Springer LNCS 6434, 70–83, 2010.
9. C.M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proc. IEEE*, **56**, 1107–1108, 1968.
10. N.P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. *Public Key Cryptography – PKC 2010*, Springer LNCS 6056, 420–443, 2010
11. N.P. Smart and F. Vercauteren. Fully Homomorphic SIMD Operations. IACR e-print 2011/133.
12. L.H. Thomas. Using a computer to solve problems in physics. *Application of Digital Computers*, 1963.
13. W.F. Trench. An algorithm for the inversion of finite Toeplitz matrices. J. SIAM, **12**, 515-522, 1964.

On CCA-Secure Somewhat Homomorphic Encryption

Jake Loftus[1], Alexander May[2], Nigel P. Smart[1], and Frederik Vercauteren[3]

[1] Dept. Computer Science,
University of Bristol,
Merchant Venturers Building, Woodland Road,
Bristol, BS8 1UB, United Kingdom.
{loftus,nigel}@cs.bris.ac.uk
[2] Horst Görtz Institute for IT-Security,
Faculty of Mathematics,
Ruhr-University Bochum, Germany
alex.may@rub.de
[3] COSIC - Electrical Engineering,
Katholieke Universiteit Leuven,
Kasteelpark Arenberg 10,
B-3001 Heverlee, Belgium.
fvercaut@esat.kuleuven.ac.be

Abstract. It is well known that any encryption scheme which supports any form of homomorphic operation cannot be secure against adaptive chosen ciphertext attacks. The question then arises as to what is the most stringent security definition which is achievable by homomorphic encryption schemes. Prior work has shown that various schemes which support a single homomorphic encryption scheme can be shown to be IND-CCA1, i.e. secure against lunchtime attacks. In this paper we extend this analysis to the recent fully homomorphic encryption scheme proposed by Gentry, as refined by Gentry, Halevi, Smart and Vercauteren. We show that the basic Gentry scheme is not IND-CCA1; indeed a trivial lunchtime attack allows one to recover the secret key. We then show that a minor modification to the variant of the somewhat homomorphic encryption scheme of Smart and Vercauteren will allow one to achieve IND-CCA1, indeed PA-1, in the standard model assuming a lattice based knowledge assumption. We also examine the security of the scheme against another security notion, namely security in the presence of ciphertext validity checking oracles; and show why CCA-like notions are important in applications in which multiple parties submit encrypted data to the "cloud" for secure processing.

1 Introduction

That some encryption schemes allow homomorphic operations, or exhibit so called *privacy homomorphisms* in the language of Rivest et. al [24], has often been considered a weakness. This is because any scheme which supports homomorphic operations is malleable, and hence is unable to achieve the de-facto security definition for encryption namely IND-CCA2. However, homomorphic encryption schemes do present a number of functional benefits. For example schemes which support a single additive homomorphic operation have been used to construct secure electronic voting schemes, e.g. [9,12].

The usefulness of schemes supporting a single homomorphic operation has led some authors to consider what security definition existing homomorphic encryption schemes meet. A natural notion to try to achieve is that of IND-CCA1, i.e. security in the presence of a lunch-time attack. Lipmaa [20] shows that the ElGamal encryption scheme is IND-CCA1 secure with respect to a hard problem which is essentially the same as the IND-CCA1 security of the ElGamal scheme; a path of work recently extended in [2] to other schemes.

A different line of work has been to examine security in the context of Plaintext Awareness, introduced by Bellare and Rogaway [5] in the random oracle model and later refined into a hierarchy of security notions (PA-0, -1 and -2) by Bellare and Palacio [4]. Intuitively a scheme is said to be PA if the only way an adversary can create a valid ciphertext is by applying encryption to a public key and a valid message. Bellare and Palacio prove that a scheme which possesses both PA-1 (resp. PA-2) and is IND-CPA, is in fact secure against IND-CCA1 (resp. IND-CCA2) attacks.

The advantage of Bellare and Palacio's work is that one works in the standard model to prove security of a scheme; the disadvantage appears to be that one needs to make a strong assumption to prove a scheme is PA-1 or PA-2. The assumption required is a so-called *knowledge assumption*. That such a strong assumption is needed should not be surprising as the PA security notions are themselves very strong. In the context of encryption schemes supporting a single homomorphic operation Bellare and Pallacio show that the Cramer-Shoup Lite scheme [10] and an ElGamal variant introduced by Damgård [11] are both PA-1, and hence IND-CCA1, assuming the standard DDH (to obtain IND-CPA security) and a Diffie–Hellman knowledge assumption (to obtain PA-1 security). Informally, the Diffie–Hellman knowledge assumption is the assumption that an algorithm can only output a Diffie–Hellman tuple if the algorithm "knows" the discrete logarithm of one-tuple member with respect to another.

Rivest et. al originally proposed homomorphic encryption schemes so as to enable arbitrary computation on encrypted data. To perform such operations one would require an encryption scheme which supports two homomorphic operations, which are "complete" in the sense of allowing arbitrary computations. Such schemes are called fully homomorphic encryption (FHE) schemes, and it was not until Gentry's breakthrough construction in 2009 [15,16] that such schemes could be constructed. Since Gentry's construction appeared a number of variants have been proposed, such as [14], as well as various simplifications [27] and improvements thereof [17]. All such schemes have been proved to be IND-CPA, i.e. secure under chosen plaintext attack.

At a high level all these constructions work in three stages: an initial *somewhat* homomorphic encryption (SHE) scheme which supports homomorphic evaluation of low degree polynomials, a process of squashing the decryption circuit and finally a bootstrapping procedure which will give fully homomorphic encryption and the evaluation of arbitrary functions on ciphertexts. In this paper we focus solely on the basic somewhat homomorphic scheme, but our attacks and analysis apply also to the extension using the bootstrapping process. Our construction of an IND-CCA1 scheme however only applies to the SHE constructions as all existing FHE constructions require public keys which already contain ciphertexts; thus with existing FHE constructions the notion

of IND-CCA1 security is redundant; although in Section 7 we present a notion of CCA embeddability which can be extended to FHE.

In this paper we consider the Smart–Vercauteren variant [27] of Gentry's scheme. In this variant there are two possible message spaces; one can either use the scheme to encrypt bits, and hence perform homomorphic operations in $\mathbb{F}_2$; or one can encrypt polynomials of degree N over $\mathbb{F}_2$. When one encrypts bits one achieves a scheme that is a specialisation of the original Gentry scheme, and it is this variant that has recently been realised by Gentry and Halevi [17]. We call this the Gentry–Halevi variant, to avoid confusion with other variants of Gentry's scheme, and we show that this scheme is not IND-CCA1 secure.

In particular in Section 4 we present a trivial complete break of the Gentry–Halevi variant scheme, in which the secret key can be recovered via a polynomial number of queries to a decryption oracle. The attack we propose works in a similar fashion to the attack of Bleichenbacher on RSA [8], in that on each successive oracle call we reduce the possible interval containing the secret key, based on the output of the oracle. Eventually the interval contains a single element, namely the secret key. Interesting all the Bleichenbacher style attacks on RSA, [8,21,26], recover a target message, and are hence strictly CCA2 attacks, whereas our attack takes no target ciphertext and recovers the key itself.

In Section 5 we go on to show that a modification of the Smart–Vercauteren SHE variant which encrypts polynomials can be shown to be PA-1, and hence is IND-CCA1. Informally we use the full Smart–Vercauteren variant to recover the random polynomial used to encrypt the plaintext polynomial in the decryption phase, and then we re-encrypt the result to check against the ciphertext. This forms a ciphertext validity check which then allows us to show PA-1 security based on a new lattice knowledge assumption. Our lattice knowledge assumption is a natural lattice based variant of the Diffie–Hellman knowledge assumption mentioned previously. In particular we assume that if an algorithm is able to output a non-lattice vector which is sufficiently close to a lattice vector then it must "know" the corresponding close lattice vector. We hope that this problem may be of independent interest in analysing other lattice based cryptographic schemes; indeed the notion is closely linked to a key "quantum" step in the results of Regev [23].

In Section 6 we examine possible extensions of the security notion for homomorphic encryption. We have remarked that a homomorphic encryption scheme (either one which supports single homomorphic operations, or a SHE/FHE scheme) cannot be IND-CCA2, but we have examples of singlely homomorphic and SHE IND-CCA1 schemes. The question then arises as to whether IND-CCA1 is the "correct" security definition, i.e. whether this is the strongest definition one can obtain for SHE schemes. In other contexts authors have considered attacks involving partial information oracles. In [13] Dent introduces the notion of a CPA+ attack, where the adversary is given access to an oracle which on input of a ciphertext outputs a single bit indicating whether the ciphertext is valid or not. Such a notion was originally introduced by Joye, Quisquater and Yung [19] in the context of attacking a variant of the EPOC-2 cipher which had been "proved" IND-CCA2. This notion was recently re-introduced under the name of a CVA (ciphertext verification) attack by Hu et al [18], in the context of symmetric en-

cryption schemes. We use the term CVA rather than CPA+ as it conveys more easily the meaning of the security notion.

Such ciphertext validity oracles are actually the key component behind the traditional application of Bleichenbacher style attacks against RSA, in that one uses the oracle to recover information about the target plaintext. We show that our SHE scheme which is IND-CCA1 is not IND-CVA, by presenting an IND-CVA attack. In particular this shows that CVA security is not implied by PA-1 security. Given PA-1 is such a strong notion this is itself interesting since it shows that CVA attacks are relatively powerful. The attack is not of the Bleichenbacher type, but is now more akin to the security reduction between search and decision LWE [25]. This attack opens up the possibility of a new SHE scheme which is also IND-CVA, a topic which we leave as an open problem; or indeed the construction of standard additive or multiplicative homomorphic schemes which are IND-CVA.

Finally, in Section 7 we consider an application area of cloud computing in which multiple players submit encrypted data to a cloud computer; which in turn will perform computations on the encrypted data. We show that such a scenario does indeed seem to require a form of IND-CCA2 protection of ciphertexts, yet still maintaining homomorphic properties. To deal with this we introduce the notion of CCA-embeddable homomorphic encryption.

2 Notation and Standard Definitions

For integers z, d reduction of z modulo d in the interval $[-d/2, d/2)$ will be denoted by $[z]_d$. For a rational number q, $\lfloor q \rceil$ will denote the rounding of q to the nearest integer, and $[q]$ denotes the (signed) distance between q and the nearest integer, i.e. $\lfloor q \rceil = q - [q]$. The notation $a \leftarrow b$ means assign the object b to a, whereas $a \leftarrow B$ for a set B means assign a uniformly at random from the set B. If B is an algorithm this means assign a with the output of B where the probability distribution is over the random coins of B.

For a polynomial $F(X) \in \mathbb{Q}[X]$ we let $\|F(X)\|_\infty$ denote the ∞-norm of the coefficient vector, i.e. the maximum coefficient in absolute value. If $F(X) \in \mathbb{Q}[X]$ then we let $\lfloor F(X) \rceil$ denote the polynomial in $\mathbb{Z}[X]$ obtained by rounding the coefficients of $F(X)$ to the nearest integer.

FULLY HOMOMORPHIC ENCRYPTION: A fully homomorphic encryption scheme is a tuple of three algorithms $\mathcal{E} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ in which the message space is a ring $(R, +, \cdot)$ and the ciphertext space is also a ring $(\mathcal{R}, \oplus, \otimes)$ such that for all messages $m_1, m_2 \in R$, and all outputs $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$, we have

$$m_1 + m_2 = \mathsf{Decrypt}(\mathsf{Encrypt}(m_1, \mathsf{pk}) \oplus \mathsf{Encrypt}(m_2, \mathsf{pk}), \mathsf{sk})$$
$$m_1 \cdot m_2 = \mathsf{Decrypt}(\mathsf{Encrypt}(m_1, \mathsf{pk}) \otimes \mathsf{Encrypt}(m_2, \mathsf{pk}), \mathsf{sk}).$$

A scheme is said to be *somewhat* homomorphic if it can deal with only a limited number of addition and multiplications before decryption fails.

SECURITY NOTIONS FOR PUBLIC KEY ENCRYPTION: Semantic security of a public key encryption scheme, whether standard, homomorphic, or fully homomorphic, is cap-

tured by the following game between a challenger and an adversary $\mathcal{A}$, running in two stages;

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$.
- $(m_0, m_1, \mathsf{St}) \leftarrow \mathcal{A}_1^{(\cdot)}(\mathsf{pk})$. /* Stage 1 */
- $b \leftarrow \{0, 1\}$.
- $c^* \leftarrow \mathsf{Encrypt}(m_b, \mathsf{pk}; r)$.
- $b' \leftarrow \mathcal{A}_2^{(\cdot)}(c^*, \mathsf{St})$. /* Stage 2 */

The adversary is said to win the game if $b = b'$, with the advantage of the adversary winning the game being defined by

$$\mathrm{Adv}^{IND-atk}_{\mathcal{A},\mathcal{E},\lambda} = |\Pr(b = b') - 1/2| .$$

A scheme is said to be IND-atk secure if no polynomial time adversary $\mathcal{A}$ can win the above game with non-negligible advantage in the security parameter λ. The precise security notion one obtains depends on the oracle access one gives the adversary in its different stages.

- If $\mathcal{A}$ has access to no oracles in either stage then atk=CPA.
- If $\mathcal{A}$ has access to a decryption oracle in stage one then atk=CCA1.
- If $\mathcal{A}$ has access to a decryption oracle in both stages then atk=CCA2, often now denoted simply CCA.
- If $\mathcal{A}$ has access to a ciphertext validity oracle in both stages, which on input of a ciphertext determines whether it would output $\perp$ or not on decryption, then atk=CVA.

LATTICES: A (full-rank) lattice is simply a discrete subgroup of $\mathbb{R}^n$ generated by n linear independent vectors, $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, called a basis. Every lattice has an infinite number of bases, with each set of basis vectors being related by a unimodular transformation matrix. If B is such a set of vectors, we write

$$L = \mathcal{L}(B) = \{\mathbf{v} \cdot B | \mathbf{v} \in \mathbb{Z}^n\}$$

to be the resulting lattice. An integer lattice is a lattice in which all the bases vectors have integer coordinates.

For any basis there is an associated fundamental parallelepiped which can be taken as $\mathcal{P}(B) = \{\sum_{i=1}^n x_i \cdot \mathbf{b}_i | x_i \in [-1/2, 1/2)\}$. The volume of this fundamental parallelepiped is given by the absolute value of the determinant of the basis matrix $\Delta = |\det(B)|$. We denote by $\lambda_\infty(L)$ the ∞-norm of a shortest vector (for the ∞-norm) in L.

3 The Smart-Vercauteren Variant of Gentry's Scheme

We will be examining variants of Gentry's SHE scheme [15], in particular three variants based on the simplification of Smart and Vercauteren [27], as optimized by Gentry and Halevi [17]. All variants make use of the same key generation procedure, parametrized

by a tuple of integers (N, t, μ); we assume there is a function mapping security parameters λ into tuples (N, t, μ). In practice N will be a power of two, t will be greater than $2^{\sqrt{N}}$ and μ will be a small integer, perhaps one.

$\mathsf{KeyGen}(1^\lambda)$
- Pick an irreducible polynomial $F \in \mathbb{Z}[X]$ of degree N.
- Pick a polynomial $G(X) \in \mathbb{Z}[X]$ of degree at most $N-1$, with coefficients bounded by t.
- $d \leftarrow \mathsf{resultant}(F, G)$.
- G is chosen such that $G(X)$ has a single unique root in common with $F(X)$ modulo d. Let α denote this root.
- $Z(X) \leftarrow d/G(X) \pmod{F(X)}$.
- $\mathsf{pk} \leftarrow (\alpha, d, \mu, F(X))$, $\mathsf{sk} \leftarrow (Z(X), G(X), d, F(X))$.

In [17] Gentry and Halevi show how to compute, for the polynomial $F(X) = X^{2^n} + 1$, the root α and the polynomial $Z(X)$ using a method based on the Fast Fourier Transform. In particular they show how this can be done for non-prime values of d (removing one of the main restrictions in the key generation method proposed in [27]).

By construction, the principal ideal $\mathfrak{g}$ generated by $G(X)$ in the number field $K = \mathbb{Z}[X]/(F(X))$ is equal to the ideal with $\mathcal{O}_K$ basis $(d, X - \alpha)$. In particular, the ideal $\mathfrak{g}$ precisely consists of all elements in $\mathbb{Z}[X]/(F(X))$ that are zero when evaluated at α modulo d. The Hermite-Normal-Form of a basis matrix of the lattice defined by the coefficient vectors of $\mathfrak{g}$ is given by

$$B = \begin{pmatrix} d & & & 0 \\ -\alpha & 1 & & \\ -\alpha^2 & & 1 & \\ \vdots & & & \ddots \\ -\alpha^{N-1} & 0 & & 1 \end{pmatrix}, \tag{1}$$

where the elements in the first column are reduced modulo d.

To aid what follows we write $Z(X) = z_0 + z_1 \cdot X + \ldots + z_{N-1} \cdot X^{N-1}$ and define

$$\delta_\infty = \sup \left\{ \frac{\|g(X) \cdot h(X) \pmod{F(X)}\|_\infty}{\|g(X)\|_\infty \cdot \|h(X)\|_\infty} : g, h \in \mathbb{Z}[X], \deg(g), \deg(h) < N \right\}.$$

For the choice $f = X^N + 1$, we have $\delta_\infty = N$. The key result to understand how the simplification of Smart and Vercauteren to Gentry's scheme works is the following lemma adapted from [27].

Lemma 1. *Let $Z(X), G(X), \alpha$ and d be as defined in the above key generation procedure. If $C(X) \in \mathbb{Z}[X]/(F(X))$ is a polynomial with $\|C(X)\|_\infty < U$ and set $c = C(\alpha) \pmod{d}$, then*

$$C(X) = c - \left\lfloor \frac{c \cdot Z(X)}{d} \right\rceil \cdot G(X) \pmod{F(X)}$$

for

$$U = \frac{d}{2 \cdot \delta_\infty \cdot \|Z(X)\|_\infty}.$$

Proof. By definition of c, we have that $c - C(X)$ is contained in the principal ideal generated by $G(X)$ and thus there exists a $q(X) \in \mathbb{Z}[X]/(F(X))$ such that $c - C(X) = q(X)G(X)$. Using $Z(X) = d/G(X) \pmod{F(X)}$, we can write

$$q(X) = \frac{cZ(X)}{d} - \frac{C(X)Z(X)}{d} .$$

Since $q(X)$ has integer coefficients, we can recover it by rounding the coefficients of the first term if the coefficients of the second term are strictly bounded by $1/2$. This shows that $C(X)$ can be recovered from c for $\|C(X)\|_\infty < d/(2 \cdot \delta_\infty \cdot \|Z(X)\|_\infty)$.

Note that the above lemma essentially states that if $\|C(X)\|_\infty < U$, then $C(X)$ is determined uniquely by its evaluation in α modulo d. Recall that any polynomial $H(X)$ of degree less than N, whose coefficient vector is in the lattice defined in equation (1), satisfies $H(\alpha) = 0 \pmod{d}$. Therefore, if $H(X) \neq 0$, the lemma implies, for such an H, that $\|H(X)\|_\infty \geq U$, and thus we conclude that $U \leq \lambda_\infty(L)$. Since the coefficient vector of $G(X)$ is clearly in the lattice L, we conclude that

$$U \leq \lambda_\infty(L) \leq \|G(X)\|_\infty .$$

Although Lemma 1 provides the maximum value of U for which ciphertexts are decryptable, we will only allow a quarter of this maximum value, i.e. $T = U/4$. As such we are guaranteed that $T \leq \lambda_\infty(L)/4$. We note that T defines the size of the circuit that the somewhat homomorphic encryption scheme can deal with. Our choice of T will become clear in Section 5.

Using the above key generation method we can define three variants of the Smart–Vercauteren variant of Gentry's scheme. The first variant is the one used in the Gentry/Halevi implementation of [17], the second is the general variant proposed by Smart and Vercauteren, whereas the third divides the decryption procedure into two steps and provides a ciphertext validity check. In later sections we shall show that the first variant is not IND-CCA1 secure, and by extension neither is the second variant. However, we will show that the third variant is indeed IND-CCA1. We will then show that the third variant is not IND-CVA secure.

Each of the following variants is only a somewhat homomorphic scheme, extending it to a fully homomorphic scheme can be performed using methods of [15,16,17].

GENTRY–HALEVI VARIANT: The plaintext space is the field $\mathbb{F}_2$. The above KeyGen algorithm is modified to only output keys for which $d \equiv 1 \pmod{2}$. This implies that at least one coefficient of $Z(X)$, say z_{i_0} will be odd. We replace $Z(X)$ in the private key with z_{i_0}, and can drop the values $G(X)$ and $F(X)$ entirely from the private key. Encryption and decryption can now be defined via the functions:

Encrypt$(m, \mathsf{pk}; r)$
- $R(X) \leftarrow \mathbb{Z}[X]$ s.t. $\|R(X)\|_\infty \leq \mu$.
- $C(X) \leftarrow m + 2 \cdot R(X)$.
- $c \leftarrow [C(\alpha)]_d$.
- Return c.

Decrypt(c, sk)
- $m \leftarrow [c \cdot z_{i_0}]_d \pmod{2}$
- Return m.

FULL-SPACE SMART–VERCAUTEREN: In this variant the plaintext space is the algebra $\mathbb{F}_2[X]/(F(X))$, where messages are given by binary polynomials of degree less than N. As such we call this the Full-Space Smart–Vercauteren system as the plaintext space is the full set of binary polynomials, with multiplication and addition defined modulo $F(X)$. We modify the above key generation algorithm so that it only outputs keys for which the polynomial $G(X)$ satisifies $G(X) \equiv 1 \pmod 2$. This results in algorithms defined by:

$\mathsf{Encrypt}(M(X), \mathsf{pk}; r)$
- $R(X) \leftarrow \mathbb{Z}[X]$ s.t. $\|R(X)\|_\infty \leq \mu$.
- $C(X) \leftarrow M(X) + 2 \cdot R(X)$.
- $c \leftarrow [C(\alpha)]_d$.
- Return c.

$\mathsf{Decrypt}(c, \mathsf{sk})$
- $C(X) \leftarrow c - \lfloor c \cdot Z(X)/d \rceil$.
- $M(X) \leftarrow C(X) \pmod 2$.
- Return $M(X)$.

That decryption works, assuming the input ciphertext corresponds to the evaluation of a polynomial with coefficients bounded by T, follows from Lemma 1 and the fact that $G(X) \equiv 1 \pmod 2$.

CCSHE: This is our ciphertext-checking SHE scheme (or ccSHE scheme for short). This is exactly like the above Full-Space Smart–Vercauteren variant in terms of key generation, but we now check the ciphertext before we output the message. Thus encryption/decryption become;

$\mathsf{Encrypt}(M(X), \mathsf{pk}; r)$
- $R(X) \leftarrow \mathbb{Z}[X]$ s.t. $\|R(X)\|_\infty \leq \mu$.
- $C(X) \leftarrow M(X) + 2 \cdot R(X)$.
- $c \leftarrow [C(\alpha)]_d$.
- Return c.

$\mathsf{Decrypt}(c, \mathsf{sk})$
- $C(X) \leftarrow c - \lfloor c \cdot Z(X)/d \rceil \cdot G(X)$.
- $C(X) \leftarrow C(X) \pmod{F(X)}$
- $c' \leftarrow [C(\alpha)]_d$.
- If $c' \neq c$ or $\|C(X)\|_\infty > T$ return $\perp$.
- $M(X) \leftarrow C(X) \pmod 2$.
- Return $M(X)$.

4 CCA1 attack on the Gentry–Halevi Variant

We construct an IND-CCA1 attacker against the above Gentry–Halevi variant. Let z be the secret key, i.e. the specific odd coefficient of $Z(X)$ chosen by the decryptor. Note that we can assume $z \in [0, d)$, since decryption in the Gentry–Halevi variant works for any secret key $z + k \cdot d$ with $k \in \mathbb{Z}$. We assume the attacker has access to a decryption oracle to which it can make polynomially many queries, $\mathcal{O}_D(c)$. On each query the oracle returns the value of $[c \cdot z]_d \pmod 2$.

In Algorithm 1 we present pseudo-code to describe how the attack proceeds. We start with an interval $[L, \ldots, U]$ which is known to contain the secret key z and in each iteration we split the interval into two halves determined by a specific ciphertext c. The choice of which sub-interval to take next depends on whether k multiples of d are sufficient to reduce $c \cdot z$ into the range $[-d/2, \ldots, d/2)$ or whether $k + 1$ multiples are required.

ANALYSIS: The core idea of the algorithm is simple: in each step we choose a "ciphertext" c such that the length of the interval for the quantity $c \cdot z$ is bounded by d. Since in

Algorithm 1: CCA1 attack on the Gentry–Halevi Variant

$L \leftarrow 0, U \leftarrow d-1$
while $U - L > 1$ **do**
  $c \leftarrow \lfloor d/(U-L) \rfloor$
  $b \leftarrow \mathcal{O}_D(c)$
  $q \leftarrow (c+b) \mod 2$
  $k \leftarrow \lfloor Lc/d + 1/2 \rfloor$
  $B \leftarrow (k+1/2)d/c$
  if $(k \bmod 2 = q)$ **then**
    $U \leftarrow \lfloor B \rfloor$
  else
    $L \leftarrow \lceil B \rceil$
return L

each step, $z \in [L, U]$, we need to take $c = \lfloor d/(U-L) \rfloor$. As such it is easy to see that $c(U-L) \leq d$.

To reduce cL, we need to subtract kd such that $-d/2 \leq cL - kd < d/2$, which shows that $k = \lfloor Lc/d + 1/2 \rfloor$. Furthermore, since the length of the interval for $c \cdot z$ is bounded by d, there will be exactly one number of the form $d/2 + id$ in $[cL, cU]$, namely $d/2 + kd$. This means that there is exactly one boundary $B = (k+1/2)d/c$ in the interval for z.

Define q as the unique integer such that $-d/2 \leq cz - qd < d/2$, then since the length of the interval for $c \cdot z$ is bounded by d, we either have $q = k$ or $q = k+1$. To distinguish between the two cases, we simply look at the output of the decryption oracle: recall that the oracle outputs $[c \cdot z]_d \pmod 2$, i.e. the bit output by the oracle is

$$b = c \cdot z - q \cdot d \pmod 2 = (c+q) \pmod 2 .$$

Therefore, $q = (b+c) \pmod 2$ which allows us to choose between the cases k and $k+1$. If $q = k \pmod 2$, then z lies in the first part $[L, \lfloor B \rfloor]$, whereas in the other case, z lies in the second part $[\lceil B \rceil, U]$.

Having proved correctness we now estimate the running time. The behaviour of the algorithm is easily seen to be as follows: in each step, we obtain a boundary B in the interval $[L, U]$ and the next interval becomes either $[L, \lfloor B \rfloor]$ or $[\lceil B \rceil, U]$. Since B can be considered random in $[L, U]$ as well as the choice of the interval, this shows that in each step, the size of the interval decreases by a factor 2 on average. In conclusion we deduce that recovering the secret key will require $O(\log d)$ calls to the oracle.

The above attack is highly efficient in practice and recovers keys in a matter of seconds for all parameter sizes in [17].

5 ccSHE is PA-1

In this section we prove that the ccSHE encryption scheme given earlier is PA-1, assuming a lattice knowledge assumption holds. We first recap on the definition of PA-1 in the standard model, and then we introduce our lattice knowledge assumption. Once this is done we present the proof.

PLAINTEXT AWARENESS – PA-1: The original intuition for the introduction of plaintext awareness was as follows - if an adversary knows the plaintext corresponding to every ciphertext it produces, then the adversary has no need for a decryption oracle and hence, PA+IND-CPA must imply IND-CCA. Unfortunately, there are subtleties in the definition for plaintext awareness, leading to three definitions, PA-0, PA-1 and PA-2. However, after suitably formalizing the definitions, PA-x plus IND-CPA implies IND-CCAx, for x = 1 and 2. In our context we are only interested in IND-CCA1 security, so we will only discuss the notion of PA-1 in this paper.

Before formalizing PA-1 it is worth outlining some of the terminology. We have a polynomial time adversary $\mathcal{A}$ called a *ciphertext creator*, that takes as input a public key and can query ciphertexts to an oracle. An algorithm $\mathcal{A}^*$ is called a *successful extractor* for $\mathcal{A}$ if it can provide responses to $\mathcal{A}$ which are computationally indistinguishable from those provided by a decryption oracle. In particular a scheme is said to be PA-1 if there exists a successful extractor for any ciphertext creator that makes a polynomial number of queries. The extractor gets the same public key as $\mathcal{A}$ and also has access to the random coins used by algorithm $\mathcal{A}$. Following [4] we define PA-1 formally as follows:

Definition 1 (PA1). *Let $\mathcal{E}$ be a public key encryption scheme and $\mathcal{A}$ be an algorithm with access to an oracle $\mathcal{O}$ taking input* pk *and returning a string. Let $\mathcal{D}$ be an algorithm that takes as input a string and returns a single bit and let $\mathcal{A}^*$ be an algorithm which takes as input a string and some state information and returns either a string or the symbol $\perp$, plus a new state. We call $\mathcal{A}$ a* ciphertext creator*, $\mathcal{A}^*$ a* PA-1-extractor*, and $\mathcal{D}$ a* distinguisher*. For security parameter λ we define the (distinguishing and extracting) experiments in Figure 1, and then define the PA-1 advantage to be*

$$\mathrm{Adv}^{PA\text{-}1}_{\mathcal{E},\mathcal{A},\mathcal{D},\mathcal{A}^*}(\lambda) = \left|\Pr(\mathrm{Exp}^{PA\text{-}1\text{-}d}_{\mathcal{E},\mathcal{A},\mathcal{D}}(\lambda)=1) - \Pr(\mathrm{Exp}^{PA\text{-}1\text{-}x}_{\mathcal{E},\mathcal{A},\mathcal{D},\mathcal{A}^*}(\lambda)=1)\right|.$$

We say $\mathcal{A}^*$ is a successful PA-1-extractor for $\mathcal{A}$, if for every polynomial time distinguisher the above advantage is negligible.

$\mathrm{Exp}^{PA\text{-}1\text{-}d}_{\mathcal{E},\mathcal{A},\mathcal{D}}(\lambda)$:
- $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$.
- $x \leftarrow \mathcal{A}^{\mathsf{Decrypt}(\cdot,\mathsf{sk})}(\mathsf{pk})$.
- $d \leftarrow \mathcal{D}(x)$.
- Return d.

$\mathrm{Exp}^{PA\text{-}1\text{-}x}_{\mathcal{E},\mathcal{A},\mathcal{A}^*}(\lambda)$:
- $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$.
- Choose coins $\mathsf{coins}[\mathcal{A}]$ (resp. $\mathsf{coins}[\mathcal{A}^*]$) for $\mathcal{A}$ (resp. $\mathcal{A}^*$).
- $\mathsf{St} \leftarrow (\mathsf{pk}, \mathsf{coins}[\mathcal{A}])$.
- $x \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{pk}; \mathsf{coins}[\mathcal{A}])$, replying to the oracle queries $\mathcal{O}(c)$ as follows:
  - $(m, \mathsf{St}) \leftarrow \mathcal{A}^*(c, \mathsf{St}; \mathsf{coins}[\mathcal{A}^*])$.
  - Return m to $\mathcal{A}$
- $d \leftarrow \mathcal{D}(x)$.
- Return d.

Fig. 1. Experiments $\mathrm{Exp}^{PA\text{-}1\text{-}d}_{\mathcal{E},\mathcal{A},\mathcal{D}}$ and $\mathrm{Exp}^{PA\text{-}1\text{-}x}_{\mathcal{E},\mathcal{A},\mathcal{A}^*}$

Note, in experiment $\mathrm{Exp}^{PA\text{-}1\text{-}d}_{\mathcal{E},\mathcal{A},\mathcal{D}}(\lambda)$ the algorithm $\mathcal{A}$'s oracle queries are responded to by the genuine decryption algorithm, whereas in $\mathrm{Exp}^{PA\text{-}1\text{-}x}_{\mathcal{E},\mathcal{A},\mathcal{A}^*}(\lambda)$ the queries are re-

sponded to by the PA-1-extractor. If $\mathcal{A}^*$ did not receive the coins $\mathsf{coins}[\mathcal{A}]$ from $\mathcal{A}$ then it would be functionally equivalent to the real decryption oracle, thus the fact that $\mathcal{A}^*$ gets access to the coins in the second experiment is crucial. Also note that the distinguisher acts independently of $\mathcal{A}^*$, and thus this is strictly stronger than having $\mathcal{A}$ decide as to whether it is interacting with an extractor or a real decryption oracle.

The intuition is that $\mathcal{A}^*$ acts as the unknowing subconscious of $\mathcal{A}$, and is able to extract knowledge about $\mathcal{A}$'s queries to its oracle. That $\mathcal{A}^*$ can obtain the underlying message captures the notion that $\mathcal{A}$ needs to know the message before it can output a valid ciphertext.

The following lemma is taken from [4] and will be used in the proof of the main theorem.

Lemma 2. *Let $\mathcal{E}$ be a public key encryption scheme. Let $\mathcal{A}$ be a polynomial-time ciphertext creator attacking $\mathcal{E}$, $\mathcal{D}$ a polynomial-time distinguisher, and $\mathcal{A}^*$ a polynomial-time PA-1-extractor. Let* DecOK *denote the event that all $\mathcal{A}^*$'s answers to $\mathcal{A}$'s queries are correct in experiment* $\mathrm{Exp}^{PA\text{-}1\text{-}x}_{\mathcal{E},\mathcal{A},\mathcal{D},\mathcal{A}^*}(\lambda)$*. Then,*

$$\Pr(\mathrm{Exp}^{PA\text{-}1\text{-}x}_{\mathcal{E},\mathcal{A},\mathcal{D},\mathcal{A}^*}(\lambda) = 1) \geq \Pr(\mathrm{Exp}^{PA\text{-}1\text{-}d}_{\mathcal{E},\mathcal{A},\mathcal{D}}(\lambda) = 1) - \Pr(\overline{\mathsf{DecOK}})$$

LATTICE KNOWLEDGE ASSUMPTION: Our knowledge assumption can be stated informally as follows: suppose there is a (probabilistic) algorithm $\mathcal{C}$ which takes as input a lattice basis of a lattice L and outputs a vector $\mathbf{c}$ suitably close to a lattice point $\mathbf{p}$, i.e. closer than $\epsilon \cdot \lambda_\infty(L)$ in the ∞-norm for a fixed $\epsilon \in (0, 1/2)$. Then there is an algorithm $\mathcal{C}^*$ which on input of $\mathbf{c}$ and the random coins of $\mathcal{C}$ outputs a close lattice vector $\mathbf{p}$, i.e. one for which $\|\mathbf{c} - \mathbf{p}\|_\infty < \epsilon \cdot \lambda_\infty(L)$. Note that the algorithm $\mathcal{C}^*$ can therefore act as a ϵ-CVP-solver for $\mathbf{c}$ in the ∞-norm, given the coins $\mathsf{coins}[\mathcal{C}]$. Again as in the PA-1 definition it is perhaps useful to think of $\mathcal{C}^*$ as the "subconscious" of $\mathcal{C}$, since $\mathcal{C}$ is capable of outputting a vector close to the lattice it must have known the close lattice vector in the first place. Formally we have:

Definition 2 (LK-ϵ). *Let ϵ be a fixed constant in the interval $(0, 1/2)$. Let $\mathcal{G}$ denote an algorithm which on input of a security parameter 1^λ outputs a lattice L given by a basis B of dimension $n = n(\lambda)$ and volume $\Delta = \Delta(\lambda)$. Let $\mathcal{C}$ be an algorithm that takes a lattice basis B as input, and has access to an oracle $\mathcal{O}$, and returns nothing. Let $\mathcal{C}^*$ denote an algorithm which takes as input a vector $\mathbf{c} \in \mathbb{R}^n$ and some state information, and returns another vector $\mathbf{p} \in \mathbb{R}^n$ plus a new state. Consider the experiment in Figure 2. The LK-ϵ advantage of $\mathcal{C}$ relative to $\mathcal{C}^*$ is defined by*

$$\mathrm{Adv}^{LK\text{-}\epsilon}_{\mathcal{G},\mathcal{C},\mathcal{C}^*}(\lambda) = \Pr[\mathrm{Exp}^{LK\text{-}\epsilon}_{\mathcal{G},\mathcal{C},\mathcal{C}^*}(\lambda) = 1].$$

We say $\mathcal{G}$ satisfies the LK-ϵ assumption, for a fixed ϵ, if for every polynomial time $\mathcal{C}$ there exists a polynomial time $\mathcal{C}^*$ such that $\mathrm{Adv}^{LK\text{-}\epsilon}_{\mathcal{G},\mathcal{C},\mathcal{C}^*}(\lambda)$ *is a negligible function of λ.*

The algorithm $\mathcal{C}$ is called an LK-ϵ adversary and $\mathcal{C}^*$ a LK-ϵ extractor. We now discuss this assumption in more detail. Notice, that for all lattices, if $\epsilon < 1/4$ then the probability of a random vector being within $\epsilon \cdot \lambda_\infty(L)$ of the lattice is bounded from above by $1/2^n$, and for lattices which are not highly orthogonal this is likely to hold for

$\mathrm{Exp}^{LK\text{-}\epsilon}_{\mathcal{G},\mathcal{C},\mathcal{C}^*}(\lambda)$:

- $B \leftarrow \mathcal{G}(1^\lambda)$.
- Choose coins $\mathsf{coins}[\mathcal{C}]$ (resp. $\mathsf{coins}[\mathcal{C}^*]$) for $\mathcal{C}$ (resp. $\mathcal{C}^*$).
- $\mathsf{St} \leftarrow (B, \mathsf{coins}[\mathcal{C}])$.
- Run $\mathcal{C}^{\mathcal{O}}(B; \mathsf{coins}[\mathcal{C}])$ until it halts, replying to the oracle queries $\mathcal{O}(\mathbf{c})$ as follows:
  - $(\mathbf{p}, \mathsf{St}) \leftarrow \mathcal{C}^*(\mathbf{c}, \mathsf{St}; \mathsf{coins}[\mathcal{C}^*])$.
  - If $\mathbf{p} \notin \mathcal{L}(B)$, return 1.
  - If $\|\mathbf{p} - \mathbf{c}\|_\infty > \epsilon \cdot \lambda_\infty(L)$, return 1.
  - Return $\mathbf{p}$ to $\mathcal{C}$.
- Return 0.

Fig. 2. Experiment $\mathrm{Exp}^{LK\text{-}\epsilon}_{\mathcal{G},\mathcal{C},\mathcal{C}^*}(\lambda)$

all ϵ up to $1/2$. Our choice of T in the ccSHE scheme as $U/4$ is to guarantee that our lattice knowledge assumption is applied with $\epsilon = 1/4$, and hence is more likely to hold.

If the query $\mathbf{c}$ which $\mathcal{C}$ asks of its oracle is within $\epsilon \cdot \lambda_\infty(L)$ of a lattice point then we require that $\mathcal{C}^*$ finds such a close lattice point. If it does not then the experiment will output 1; and the assumption is that this happens with negligible probability.

Notice that if $\mathcal{C}$ asks its oracle a query of a vector which is not within $\epsilon \cdot \lambda_\infty(L)$ of a lattice point then the algorithm $\mathcal{C}^*$ may do whatever it wants. However, to determine this condition within the experiment we require that the environment running the experiment is all powerful, in particular, that it can compute $\lambda_\infty(L)$ and decide whether a vector is close enough to the lattice. Thus our experiment, but not algorithms $\mathcal{C}$ and $\mathcal{C}^*$, is assumed to be information theoretic. This might seem strange at first sight but is akin to a similarly powerful game experiment in the strong security model for certificateless encryption [1], or the definition of insider unforgeable signcryption in [3].

For certain input bases, e.g. reduced ones or ones of small dimension, an algorithm $\mathcal{C}^*$ can be constructed by standard algorithms to solve the CVP problem. This does not contradict our assumption, since $\mathcal{C}$ would also be able to apply such an algorithm and hence "know" the close lattice point. Our assumption is that when this is not true, the only way $\mathcal{C}$ could generate a close lattice point (for small enough values of ϵ) is by computing $\mathbf{x} \in \mathbb{Z}^n$ and perturbing the vector $\mathbf{x} \cdot B$.

MAIN THEOREM:

Theorem 1. *Let $\mathcal{G}$ denote the lattice basis generator induced from the* KeyGen *algorithm of the ccSHE scheme, i.e. for a given security parameter 1^λ, run* $\mathsf{KeyGen}(1^\lambda)$ *to obtain* $\mathsf{pk} = (\alpha, d, \mu, F(X))$ *and* $\mathsf{sk} = (Z(X), G(X), d, F(X))$*, and generate the lattice basis B as in equation* (1)*. Then, if $\mathcal{G}$ satisfies the LK-ϵ assumption for $\epsilon = 1/4$ then the ccSHE scheme is PA-1.*

Proof. Let $\mathcal{A}$ be a polynomial-time ciphertext creator attacking the ccSHE scheme, then we show how to construct a polynomial time PA1-extractor $\mathcal{A}^*$. The creator $\mathcal{A}$ takes as input the public key $\mathsf{pk} = (\alpha, d, \mu, F(X))$ and random coins $\mathsf{coins}[\mathcal{A}]$ and returns an integer as the candidate ciphertext. To define $\mathcal{A}^*$, we will exploit $\mathcal{A}$ to build a polynomial-time LK-ϵ adversary $\mathcal{C}$ attacking the generator $\mathcal{G}$. By the LK-ϵ assumption there exists a polynomial-time LK-ϵ extractor $\mathcal{C}^*$, that will serve as the main building

block for the PA1-extractor $\mathcal{A}^*$. The description of the LK-ϵ adversary $\mathcal{C}$ is given in Figure 3 and the description of the PA-1-extractor $\mathcal{A}^*$ is given in Figure 4.

LK-ϵ adversary $\mathcal{C}^{\mathcal{O}}(B; \mathsf{coins}[\mathcal{C}])$

- Let $d = B[0][0]$ and $\alpha = -B[1][0]$
- Parse $\mathsf{coins}[\mathcal{C}]$ as $\mu||F(X)||\mathsf{coins}[\mathcal{A}]$
- Run $\mathcal{A}$ on input $(\alpha, d, \mu, F(X))$ and coins $\mathsf{coins}[\mathcal{A}]$ until it halts, replying to its oracle queries as follows:
  - If $\mathcal{A}$ makes a query with input c, then
  - Submit $(c, 0, 0, \ldots, 0)$ to $\mathcal{O}$ and let $\mathbf{p}$ denote the response
  - Let $\mathbf{c} = (c, 0, \ldots, 0) - \mathbf{p}$, and $C(X) = \sum_{i=0}^{N-1} \mathbf{c}_i X^i$
  - Let $c' = [C(\alpha)]_d$
  - If $c' \neq c$ or $\|C(X)\|_\infty \geq T$, then $M(X) \leftarrow \perp$, else $M(X) \leftarrow C(X) \pmod 2$
  - Return $M(X)$ to $\mathcal{A}$ as the oracle response.
- Halt

Fig. 3. LK-ϵ adversary

PA-1-extractor $\mathcal{A}^*(c, \mathsf{St}[\mathcal{A}^*]; \mathsf{coins}[\mathcal{A}^*])$

- If $\mathsf{St}[\mathcal{A}^*]$ is initial state then
  - parse $\mathsf{coins}[\mathcal{A}^*]$ as $(\alpha, d, \mu, F(X))||\mathsf{coins}[\mathcal{A}]$
  - $\mathsf{St}[C^*] \leftarrow (\alpha, d, \mu, F(X))||\mathsf{coins}[\mathcal{A}]$
  - else parse $\mathsf{coins}[\mathcal{A}^*]$ as $(\alpha, d, \mu, F(X))||\mathsf{St}[\mathcal{C}^*]$
- $(\mathbf{p}, \mathsf{St}[\mathcal{C}^*]) \leftarrow \mathcal{C}^*((c, 0, \ldots, 0), \mathsf{St}[\mathcal{C}^*]; \mathsf{coins}[\mathcal{A}^*])$
- Let $\mathbf{c} = (c, 0, \ldots, 0) - \mathbf{p}$, and $C(X) = \sum_{i=0}^{N-1} \mathbf{c}_i X^i$
- Let $c' = [C(\alpha)]_d$
- If $c' \neq c$ or $\|C(X)\|_\infty \geq T$, then $M(X) \leftarrow \perp$, else $M(X) \leftarrow C(X) \pmod 2$
- $\mathsf{St}[\mathcal{A}^*] \leftarrow (\alpha, d, \mu, F(X))||\mathsf{St}[C^*]$
- Return $(M(X), \mathsf{St}[\mathcal{A}^*])$.

Fig. 4. PA-1-extractor

We first show that $\mathcal{A}^*$ is a successful PA-1-extractor for $\mathcal{A}$. In particular, let DecOK denote the event that all $\mathcal{A}^*$'s answers to $\mathcal{A}$'s queries are correct in $\mathrm{Exp}^{PA\text{-}1\text{-}x}_{\mathrm{ccSHE},\mathcal{A},\mathcal{D},\mathcal{A}^*}(\lambda)$, then we have that $\Pr(\overline{\mathsf{DecOK}}) \leq \mathrm{Adv}^{LK\text{-}\epsilon}_{\mathcal{G},\mathcal{C},\mathcal{C}^*}(\lambda)$.

We first consider the case that c is a valid ciphertext, i.e. a ciphertext such that $\mathsf{Decrypt}(c, \mathsf{sk}) \neq \perp$, then by definition of Decrypt in the ccSHE scheme there exists a $C(x)$ such that $c = [C(\alpha)]_d$ and $\|C(X)\|_\infty \leq T$. Let $\mathbf{p}'$ be the coefficient vector of $c - C(X)$, then by definition of c, we have that $\mathbf{p}'$ is a lattice vector that is within distance T of the vector $(c, 0, \ldots, 0)$. Furthermore, since $T \leq \lambda_\infty(L)/4$, the vector $\mathbf{p}'$ is the *unique* vector with this property. Let $\mathbf{p}$ be the vector returned by $\mathcal{C}^*$ and assume that $\mathbf{p}$ passes the test $\|(c, 0, \ldots, 0) - \mathbf{p}\|_\infty \leq T$, then we conclude that $\mathbf{p} = \mathbf{p}'$. This shows that if c is a valid ciphertext, it will be decrypted correctly by $\mathcal{A}^*$.

When c is an invalid ciphertext then the real decryption oracle will always output $\perp$, and it can be easily seen that our PA-1 extractor $\mathcal{A}^*$ will also output $\perp$. Thus in the case of an invalid ciphertext the adversary $\mathcal{A}$ cannot tell the two oracles apart. The theorem now follows from combining the inequality $\Pr(\overline{\mathsf{DecOK}}) \leq \mathrm{Adv}^{LK\text{-}\epsilon}_{\mathcal{G},\mathcal{C},\mathcal{C}^*}(\lambda)$ with Lemma 2 as follows:

$$\begin{aligned}\mathrm{Adv}^{PA\text{-}1}_{\mathcal{E},\mathcal{A},\mathcal{D},\mathcal{A}^*}(\lambda) &= \Pr(\mathrm{Exp}^{PA\text{-}1\text{-}d}_{\mathcal{E},\mathcal{A},\mathcal{D}}(\lambda)=1) - \Pr(\mathrm{Exp}^{PA\text{-}1\text{-}x}_{\mathcal{E},\mathcal{A},\mathcal{D},\mathcal{A}^*}(\lambda)=1)\\ &\leq \Pr(\mathrm{Exp}^{PA\text{-}1\text{-}d}_{\mathcal{E},\mathcal{A},\mathcal{D}}(\lambda)=1) - \Pr(\mathrm{Exp}^{PA\text{-}1\text{-}d}_{\mathcal{E},\mathcal{A},\mathcal{D}}(\lambda)=1) + \Pr(\overline{\mathsf{DecOK}})\\ &\leq \mathrm{Adv}^{LK\text{-}\epsilon}_{\mathcal{G},\mathcal{C},\mathcal{C}^*}(\lambda)\,.\end{aligned}$$

6 ccSHE is not secure in the presence of a CVA attack

We now show that our ccSHE scheme is not secure when the attacker, after being given the target ciphertext c^*, is given access to an oracle $\mathcal{O}_{\mathrm{CVA}}(c)$ which returns 1 if c is a valid ciphertext (i.e. the decryption algorithm would output a message), and which returns 0 if it is invalid (i.e. the decryption algorithm would output $\perp$). Such an "oracle" can often be obtained in the real world by the attacker observing the behaviour of a party who is fed ciphertexts of the attackers choosing. Since a CVA attack is strictly weaker than a IND-CCA2 attack it is an interesting open (and practical) question as to whether an FHE scheme can be CVA secure.

We now show that the ccSHE scheme is not CVA secure, by presenting a relatively trivial attack: Suppose the adversary is given a target ciphertext c^* associated with a hidden message m^*. Using the method in Algorithm 2 it is easy to determine the message using access to $\mathcal{O}_{\mathrm{CVA}}(c)$. Basically, we add on multiples of α^i to the ciphertext until it does not decrypt; this allows us to perform a binary search on the i-th coefficient of $C(X)$, since we know the bound T on the coefficients of $C(X)$.

Algorithm 2: CVA attack on ccSHE

$C(X) \leftarrow 0$
for i **from** 0 **upto** $N-1$ **do**
  $L \leftarrow -T+1, U \leftarrow T-1$
  while $U \neq L$ **do**
    $M \leftarrow \lceil (U+L)/2 \rceil$.
    $c \leftarrow [-c^* + (M+T-1)\cdot \alpha^i]_d$.
    if $\mathcal{O}_{\mathrm{CVA}}(c) = 1$ **then**
      $L \leftarrow M$.
    else
      $U \leftarrow M-1$.
  $C(X) \leftarrow C(X) + U \cdot X^i$.
$m^* \leftarrow C(X) \pmod 2$
return m^*

If c_i is the ith coefficient of the actual $C(X)$ underlying the target ciphertext c^*, then the ith coefficient of the polynomial underlying ciphertext c being passed to the

$\mathcal{O}_{\mathrm{CVA}}$ oracle is given by $M + T - 1 - c_i$. When $M \leq c_i$ this coefficient is less than T and so the oracle will return 1, however when $M > c_i$ the coefficient is greater than or equal T and hence the oracle will return 0. Thus we can divide the interval for c_i in two depending on the outcome of the test.

It is obvious that the complexity of the attack is $O(N \cdot \log_2 T)$. Since, for the recommended parameters in the key generation method, N and $\log_2 T$ are polynomial functions of the security parameter, we obtain a polynomial time attack.

7 CCA2 Somewhat Homomorphic Encryption?

In this section we deal with an additional issue related to CCA security of somewhat homomorphic encryption schemes. Consider the following scenario: three parties wish to use SHE to compute some information about some data they posses. Suppose the three pieces of data are m_1, m_2 and m_3. The parties encrypt these messages with the SHE scheme to obtain ciphertexts c_1, c_2 and c_3. These are then passed to a third party who computes, via the SHE properties, the required function. The resulting ciphertext is passed to an "Opener" who then decrypts the output and passes the computed value back to the three parties. As such we are using SHE to perform a form of multi-party computation, using SHE to perform the computation and a special third party, called an Opener, to produce the final result.

Consider the above scenario in which the messages lie in $\{0, 1\}$ and the function to be computed is the majority function. Now assume that the third party and the protocol are not synchronous. In such a situation the third party may be able to make a copy of the first party's ciphertext and submit it as his own. In such a situation the third party forces the above protocol to produce an output equal to the first party's input; thus security of the first party's input is lost. This example may seem a little contrived but it is, in essence, the basis of the recent attack by Smyth and Cortier [28] on the Helios voting system; recall Helios is a voting system based on homomorphic (but not fully homomorphic) encryption.

An obvious defence against the above attack would be to disallow input ciphertexts from one party, which are identical to another party's. However, this does not preclude a party from using malleability of the underlying SHE scheme to produce a ciphertext c_3, such that $c_3 \neq c_1$, but $\mathsf{Decrypt}(c_1, \mathsf{sk}) = \mathsf{Decrypt}(c_3, \mathsf{sk})$. Hence, we need to preclude (at least) forms of benign malleability, but to do so would contradict the fact that we require a fully homomorphic encryption scheme.

To get around this problem we introduce the notion of *CCA-embeddable homomorphic encryption*. Informally this is an IND-CCA2 public key encryption scheme $\mathcal{E}$, for which given a ciphertext c one can publicly extract an equivalent ciphertext c' for an IND-CPA homomorphic encryption scheme $\mathcal{E}'$. More formally

Definition 3. *An IND-CPA homomorphic (possibly fully homomorphic) public key encryption scheme* $\mathcal{E}' = (\mathsf{KeyGen}', \mathsf{Encrypt}', \mathsf{Decrypt}')$ *is said to be* CCA-embeddable *if there is an IND-CCA encryption scheme* $\mathcal{E} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ *and an algorithm* Extract *such that*

- KeyGen *produces two secret keys* $\mathsf{sk}', \mathsf{sk}''$*, where* sk' *is in the keyspace of* $\mathcal{E}'$*.*

- $\mathsf{Decrypt}'(\mathsf{Extract}(\mathsf{Encrypt}(m, pk), \mathsf{sk}''), \mathsf{sk}') = m$.
- *The ciphertext validity check for $\mathcal{E}$ is computable using only the secret key* sk''.
- *CCA1 security of $\mathcal{E}'$ is not compromised by leakage of* sk''.

As a simple example, for standard homomorphic encryption, is that ElGamal is CCA-embeddable into the Cramer–Shoup encryption scheme [10]. We note that this notion of CCA-embeddable encryption was independently arrived at by [7] for standard (singularly) homomorphic encryption in the context of providing a defence against the earlier mentioned attack on Helios. See [7] for a more complete discussion of the concept.

As a proof of concept for somewhat homomorphic encryption schemes we show that, in the random oracle model, the somewhat homomorphic encryption schemes considered in this paper are CCA-embeddable. We do this by utilizing the Naor–Yung paradigm [22] for constructing IND-CCA encryption schemes, and the zero-knowledge proofs of knowledge for semi-homomorphic schemes considered in [6]. Note that our construction is inefficient; we leave it as an open problem as to whether more specific constructions can be provided for the specific SHE schemes considered in this paper.

CONSTRUCTION: Given an SHE scheme $\mathcal{E}' = (\mathsf{KeyGen}', \mathsf{Encrypt}', \mathsf{Decrypt}')$ we construct the scheme $\mathcal{E} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ into which $\mathcal{E}'$ embeds as follows, where $\mathsf{NIZKPoK} = (\mathsf{Prove}, \mathsf{Verify})$ is a suitable non-malleable non-interactive zero-knowledge proof of knowledge of equality of two plaintexts:

$\mathsf{KeyGen}(1^\lambda)$
- $(\mathsf{pk}'_1, \mathsf{sk}'_1) \leftarrow \mathsf{KeyGen}'(1^\lambda)$.
- $(\mathsf{pk}'_2, \mathsf{sk}'_2) \leftarrow \mathsf{KeyGen}'(1^\lambda)$.
- $\mathsf{pk} \leftarrow (\mathsf{pk}'_1, \mathsf{pk}'_2), \mathsf{sk} \leftarrow (\mathsf{sk}'_1, \mathsf{sk}'_2)$.
- Return $(\mathsf{pk}, \mathsf{sk})$.

$\mathsf{Encrypt}(m, \mathsf{pk}; r)$
- $c'_1 \leftarrow \mathsf{Encrypt}'(m, \mathsf{pk}'_1; r'_1)$.
- $c'_2 \leftarrow \mathsf{Encrypt}'(m, \mathsf{pk}'_2; r'_2)$.
- $\Sigma \leftarrow \mathsf{Prove}(c_1, c_2; m, r'_1, r'_2)$.
- $c \leftarrow (c'_1, c'_2, \Sigma)$.
- Return c.

$\mathsf{Extract}(c)$
- Parse c as (c'_1, c'_2, Σ).
- Return c'_1.

$\mathsf{Decrypt}(c, \mathsf{sk})$
- Parse c as (c'_1, c'_2, Σ).
- If $\mathsf{Verify}(\Sigma, c'_1, c'_2) = 0$ return $\perp$.
- $m \leftarrow \mathsf{Decrypt}'(c'_1, \mathsf{sk}'_1)$.
- Return m.

All that remains is to describe how to instantiate the NIZKPoK. We do this using the Fiat–Shamir heuristic applied to the Sigma-protocol in Figure 5. The protocol is derived from the same principles as those in [6], and security (completeness, soundness and zero-knowledge) can be proved in an almost identical way to that in [6]. The main difference being that we need an adjustment to be made to the response part of the protocol to deal with the message space being defined modulo two. We give the Sigma protocol in the simplified case of application to the Gentry–Halevi variant, where the message space is equal to $\{0, 1\}$. Generalising the protocol to the Full Space Smart–Vercauteren variant requires a more complex "adjustment" to the values of t_1 and t_2 in the protocol. Notice that the soundness error in the following protocol is only $1/2$, thus we need to repeat the protocol a number of times to obtain negligible soundness error which leads to a loss of efficiency.

Prover		Verifier
$c_1 = \mathsf{Encrypt}'(m, \mathsf{pk}'_1; r'_1)$		
$c_2 = \mathsf{Encrypt}'(m, \mathsf{pk}'_2; r'_2)$		c_1, c_2
$y \leftarrow \{0,1\}$		
$a_1 \leftarrow \mathsf{Encrypt}'(y, \mathsf{pk}'_1; s'_1)$		
$a_2 \leftarrow \mathsf{Encrypt}'(y, \mathsf{pk}'_2; s'_2)$	$\xrightarrow{a_1, a_2}$	
	$\xleftarrow{e}$	$e \leftarrow \{0,1\}$
$z \leftarrow y \oplus e \cdot m$		
$t_1 \leftarrow s_1 + e \cdot r_1 + e \cdot y \cdot m$		
$t_2 \leftarrow s_2 + e \cdot r_2 + e \cdot y \cdot m$	$\xrightarrow{z, t_1, t_2}$	Accept if and only if
		$\mathsf{Encrypt}'(z, pk_1; t_1) = a_1 + e \cdot c_1$
		$\mathsf{Encrypt}'(z, pk_2; t_2) = a_2 + e \cdot c_2$.

Fig. 5. ZKPoK of equality of two plaintexts

8 Acknowledgements

All authors were partially supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II. The first author was also partially funded by EPSRC and Trend Micro. The third author was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079, and by a Royal Society Wolfson Merit Award. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA), AFRL, the U.S. Government, the European Commission or EPSRC.

References

1. S.S. Al-Riyami and K.G. Patterson. Certificateless public key cryptography. In *Advances in Cryptology – ASIACRYPT 2003*, Springer LNCS 2894, 452–473, 2003.
2. F. Armknecht, A. Peter and S. Katzenbeisser. A cleaner view on IND-CCA1 secure homomorphic encryption using SOAP. IACR e-print 2010/501, `http://eprint.iacr.org/2010/501`, 2010.
3. J. Baek, R. Steinfeld and Y. Zheng. Formal proofs for the security of signcryption. *Journal of Cryptology*, **20(2)**, 203–235, 2007.
4. M. Bellare and A. Palacio. Towards Plaintext-Aware Public-Key Encryption without Random Oracles. In *Advances in Cryptology – ASIACRYPT 2004*, Springer LNCS 3329, 37-52, 2004.
5. M. Bellare and P. Rogaway. Optimal Asymmetric Encryption. In *Advances in Cryptology – EUROCRYPT'94*, Springer LNCS 950, 92-111, 1994.
6. R. Bendlin, I. Damgård, C. Orlandi and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology – EUROCRYPT 2011*, Springer LNCS 6632, 169–188, 2011.

7. D. Bernhard, V. Cortier, O. Pereira, B. Smyth and B. Warinschi. Adapting Helios for provable ballot privacy. To appear *ESORICS 2011*.
8. D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1 In *Advances in Cryptology – CRYPTO '98*, Springer LNCS 1462, 1–12,1998.
9. R. Cramer, R. Gennaro and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Advances in Cryptology – EUROCRYPT '97*, Springer LNCS 1233, 103–118, 1997.
10. R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *Advances in Cryptology – CRYPTO '98*, Springer LNCS 1462, 13–25, 1998.
11. I. Damgård Towards practical public-key schemes secure against chosen ciphertext attacks. In *Advances in Cryptology – CRYPTO '91*, Springer LNCS 576, 1991.
12. I. Damgård, J. Groth and G. Salomonsen. The theory and implementation of an electronic voting system. In *Secure Electronic Voting*, Kluwer Academic Publishers, 77–99, 2002.
13. A. Dent. A designer's guide to KEMs. In *Coding and Cryptography 2003*, Springer LNCS 2898, 133–151, 2003.
14. M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in Cryptology – EUROCRYPT 2010*, Springer LNCS 6110, 24–43, 2010.
15. C. Gentry. Fully homomorphic encryption using ideal lattices. In *Symposium on Theory of Computing – STOC 2009*, ACM, 169–178, 2009.
16. C. Gentry. A fully homomorphic encryption scheme. PhD, Stanford University, 2009.
17. C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *Advances in Cryptology – EUROCRYPT 2011*, Springer LNCS, 2011.
18. Z.-Y. Hu, F.-C. Sun and J.-C. Jiang. Ciphertext verification security of symmetric encryption schemes. *Science in China Series F*, **52(9)**, 1617–1631, 2009.
19. M. Joye, J. Quisquater, and M. Yung. On the power of misbehaving adversaries and security analysis of the original EPOC. In *Topics in Cryptography – CT-RSA 2001*, Springer LNCS 2020, 208–222, 2001.
20. H. Lipmaa. On the CCA1-security of ElGamal and Damgård's ElGamal. In *Information Security and Cryptology – INSCRYPT 2010*, Springer LNCS 6584, 18–35, 2010.
21. J. Manger. A chosen ciphertext attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as standardized in PKCS # 1 v2.0 In *Advances in Cryptology – CRYPTO '01*, Springer LNCS 2139, 230–238, 2001.
22. M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Symposium on Theory of Computing – STOC 1990*, ACM, 427–437, 1990.
23. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Symposium on Theory of Computing – STOC 2005*, ACM, 84–93, 2005.
24. R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, 169–177, 1978.
25. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal ACM*, **56(6)**, 1–40, 2009.
26. N.P. Smart. Breaking RSA-based PIN encryption with thirty ciphertext validity queries. In *Topics in Cryptology – CT-RSA 2010*, Springer LNCS 5985, 15-25, 2010.
27. N.P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography – PKC 2010*, Springer LNCS 6056, 420–443, 2010
28. B. Smyth and V. Cortier. Attacking and fixing Helios: An analysis of ballot secrecy. To appear *IEEE Computer Security Foundations Symposium – CSF 2011*.

Fully Homomorphic Encryption with Polylog Overhead

C. Gentry[1], S. Halevi[1], and N.P. Smart[2]

[1] IBM T.J. Watson Research Center,
Yorktown Heights, New York, U.S.A.
[2] Dept. Computer Science, University of Bristol,
Bristol, United Kingdom.

Abstract. We show that homomorphic evaluation of (wide enough) arithmetic circuits can be accomplished with only polylogarithmic overhead. Namely, we present a construction of fully homomorphic encryption (FHE) schemes that for security parameter λ can evaluate any width-$\Omega(\lambda)$ circuit with t gates in time $t \cdot \text{polylog}(\lambda)$.

To get low overhead, we use the recent batch homomorphic evaluation techniques of Smart-Vercauteren and Brakerski-Gentry-Vaikuntanathan, who showed that homomorphic operations can be applied to "packed" ciphertexts that encrypt vectors of plaintext elements. In this work, we introduce permuting/routing techniques to move plaintext elements across these vectors efficiently. Hence, we are able to implement general arithmetic circuit in a batched fashion without ever needing to "unpack" the plaintext vectors.

We also introduce some other optimizations that can speed up homomorphic evaluation in certain cases. For example, we show how to use the Frobenius map to raise plaintext elements to powers of p at the "cost" of a linear operation.

Keywords. Homomorphic encryption, Bootstrapping, Batching, Automorphism, Galois group, Permutation network.

Acknowledgments. The first and second authors are sponsored by DARPA and ONR under agreement number N00014-11C-0390. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

The third author is sponsored by DARPA and AFRL under agreement number FA8750-11-2-0079. The same disclaimers as above apply. He is also supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant COED–EP/I03126X, and by a Royal Society Wolfson Merit Award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the European Commission or EPSRC.

Table of Contents

1 Introduction

Fully homomorphic encryption (FHE) [16, 9, 8] allows a worker to perform arbitrarily-complex dynamically-chosen computations on encrypted data, despite not having the secret decryption key. Processing encrypted data homomorphically requires more computation than processing the data unencrypted. But how much more? What is the *overhead*, the ratio of encrypted computation complexity to unencrypted computation complexity (using a circuit model of computation)? Here, under the ring-LWE assumption, we show that the overhead can be made as low as *polylogarithmic* in the security parameter.

We accomplish this by *packing* many plaintexts into each ciphertext; each ciphertext has $\tilde{\Omega}(\lambda)$ "plaintext slots". Then, we describe a complete set of operations – Add, Mult and Permute – that allows us to evaluate arbitrary circuits *while keeping the ciphertexts packed*. Batch Add and Mult have been done before [18], and follow easily from the Chinese Remainder Theorem within our underlying polynomial ring. Here we introduce the operation Permute, that allows us to homomorphically move data between the plaintext slots, show how to realize it from our underlying algebra, and how to use it to evaluate arbitrary circuits.

Our approach begins with the observation [3, 18] that we can use an automorphism group $\mathcal{H}$ associated to our underlying ring to "rotate" or "re-align" the contents of the plaintext slots. (These automorphisms were used in a somewhat similar manner by Lyubashevsky et al. [15] in their proof of the pseudorandomness of RLWE.) While $\mathcal{H}$ alone enables only a few permutations (e.g., "rotations"), we show that any permutation can be constructed as a log-depth permutation network, where each level consists of a constant number of "rotations", batch-additions and batch-multiplications. Our method works when the underlying ring has an associated automorphism group $\mathcal{H}$ which is abelian and sharply transitive, a condition that we prove always holds for our scheme's parameters.

Ultimately, the Add, Mult and Permute operations can all be accomplished with $\tilde{O}(\lambda)$ computation by building on the recent Brakerski-Gentry-Vaikuntanathan (BGV) "FHE without bootstrapping" scheme [3], which builds on prior work by Brakerski and Vaikuntanathan and others [5, 4, 12]. Thus, we obtain an FHE scheme that can evaluate any circuit that has $\Omega(\lambda)$ average width with only polylog(λ) overhead. For comparison, the smallest overhead for FHE was $\tilde{O}(\lambda^{3.5})$ [19] until BGV recently reduced it to $\tilde{O}(\lambda)$ [3].[3]

In addition to their essential role in letting us move data across plaintext slots, ring automorphisms turn out to have interesting secondary consequences: they also enable more nimble manipulation of data *within* plaintext slots. Specifically, in some cases we can use them to raise the packed plaintext elements to a high power with hardly any increase in the noise magnitude of the ciphertext! In practice, this could permit evaluation of high-degree circuits without resorting to bootstrapping, in applications such as computing AES. See Appendix A.3.

1.1 Packing Plaintexts and Batched Homomorphic Computation

Smart and Vercauteren [17, 18] were the first to observe that, by an application the Chinese Remainder Theorem to number fields, the plaintext space of some previous FHE schemes can be partitioned into a vector of "plaintext slots", and that a single homomorphic Add or Mult of a pair of ciphertexts implicitly adds or multiplies (component-wise) the entire plaintext vectors. Each plaintext slot is defined to hold an element in some finite field $\mathbb{K}_n = \mathbb{F}_{p^n}$, and, abstractly, if one has two ciphertexts that hold (encrypt) messages $m_0, \ldots, m_{\ell-1} \in \mathbb{K}_n^\ell$ and $m'_0, \ldots, m'_{\ell-1} \in \mathbb{K}_n^\ell$ respectively in plaintext slots $0, \ldots, \ell-1$, applying ℓ-Add to the two ciphertexts gives a new ciphertext that holds $m_0 + m'_0, \ldots, m_{\ell-1} + m'_{\ell-1}$ and applying ℓ-Mult gives a new ciphertext that holds $m_0 \cdot m'_0, \ldots, m_{\ell-1} \cdot m'_{\ell-1}$. Smart and Vercauteren used this observation for *batch* (or SIMD [11]) homomorphic

[3] However, the polylog factors in our new scheme are rather large. It remains to be seen how much of an improvement this approach yields in practice, as compared to the $\tilde{O}(\lambda^{3.5})$ approach implemented in [10, 19].

operations. That is, they show how to evaluate a function f homomorphically ℓ times in parallel on ℓ different inputs, with approximately the same cost that it takes to evaluate the function once without batching.

Here is a taste of how these separate plaintext slots are constructed algebraically. As an example, for the ring-LWE-based scheme, suppose we use the polynomial ring $\mathbb{A} = \mathbb{Z}[x]/(x^\ell + 1)$ where ℓ is a power of 2. Ciphertexts are elements of $\mathbb{A}_q^2$ where (as in in [3]) q has only polylog(λ) bits. The "aggregate" plaintext space is $\mathbb{A}_p$ (that is, ring elements taken modulo p) for some small prime $p = 1 \bmod 2\ell$. Any prime $p = 1 \bmod 2\ell$ *splits* over the field associated to this ring – that is, in $\mathbb{A}$, the ideal generated by p is the product of ℓ ideals $\{\mathfrak{p}_i\}$ each of norm p – and therefore $\mathbb{A}_p \equiv \mathbb{A}_{\mathfrak{p}_0} \times \cdots \times \mathbb{A}_{\mathfrak{p}_{\ell-1}}$. Consequently, using the Chinese remainder theorem, we can encode ℓ independent mod-p plaintexts $m_0, \ldots, m_{\ell-1} \in \{0, \ldots, p-1\}$ as the unique element in $\mathbb{A}_p$ that is in all of the cosets $m_i + \mathfrak{p}_i$. Thus, in a single ciphertext, we may have ℓ independent plaintext "slots".

In this work, we often use ℓ-Add and ℓ-Mult to efficiently implement a Select operation: Given an index set I we can construct a vector $\mathbf{v}_I$ of "select bits" $(v_0, \ldots, v_{\ell-1})$, such that $v_i = 1$ if $i \in I$ and $v_i = 0$ otherwise. Then element-wise multiplication of a packed ciphertext $\mathbf{c}$ with the select vector $\mathbf{v}$ results in a new ciphertext that contains only the plaintext element in the slots corresponding to I, and zero elsewhere. Moreover, by generating two complementing select vectors $\mathbf{v}_I$ and $\mathbf{v}_{\bar{I}}$ we can mix-and-match the slots from two packed ciphertexts $\mathbf{c}_1$ and $\mathbf{c}_2$: Setting $\mathbf{c} = (\mathbf{v}_I \times \mathbf{c}_1) + (\mathbf{v}_{\bar{I}} \times \mathbf{c}_2)$, we pack into $\mathbf{c}$ the slots from $\mathbf{c}_1$ at indexes from I and the slots from $\mathbf{c}_2$ elsewhere.

While batching is useful in many setting, it does not, by itself, yield low-overhead homomorphic computation in general, as it does not help us to reduce the overhead of computing a complicated function just once. Just as in normal program execution of SIMD instructions (e.g., the SSE instructions on x86), one needs a method of moving data between slots in each SIMD word.

1.2 Permuting Plaintexts Within the Plaintext Slots

To reduce the overhead of homomorphic computation *in general*, we need a *complete* set of operations over *packed vectors of plaintexts*. The approach above allows us to add or multiply messages that are in the same plaintext slot, but what if we want to add the content of the i-th slot in one ciphertext to the content of the j-th slot of another ciphertext, for $i \neq j$? We can "unpack" the slots into separate ciphertexts (say, using homomorphic decryption[4] [8, 9]), but there is little hope that this approach could yield very efficient FHE. Instead, we complement ℓ-Add and ℓ-Mult with an operation ℓ-Permute to move data efficiently across slots within a a given ciphertext, and efficient procedures to clone slots from a packed ciphertext and move them around to other packed ciphertexts.

Brakerski, Gentry, and Vaikuntanathan [3] observed that for certain parameter settings, one can use *automorphisms* associated with the algebraic ring $\mathbb{A}$ to "rotate" all of plaintext spaces simultaneously, sort of like turning a dial on a safe. That is, one can transform a ciphertext that holds $m_0, m_1, \ldots, m_{\ell-1}$ in its ℓ slots into another ciphertext that holds $m_i, m_{i+1}, \ldots, m_{i+\ell-1}$ (for an arbitrary given i, index arithmetic mod ℓ), and this rotation operation takes time quasi-linear in the ciphertext size, which is quasi-linear in the security parameter. They used this tool to construct Pack and Unpack algorithms whereby separate ciphertexts could be aggregated (packed) into a single ciphertext with packed plaintexts before applying bootstrapping (and then the refreshed ciphertext would be unpacked), thereby lowering the amortized cost of bootstrapping.

We exploit these automorphisms more fully, using the basic rotations that the automorphisms give us to construct *permutation networks* that can permute data in the plaintext slots arbitrarily. We also extend the application of the automorphisms to more general underlying rings, beyond the specific parameter settings considered in prior work [5, 4, 3]. This lets us devise low-overhead homomorphic schemes for arithmetic circuits over essentially any small finite field $\mathbb{F}_{p^n}$.

[4] This is the approach suggested in [18] for Gentry's original FHE scheme.

Our efficient implementation of Permute, described in Section 3, uses the Beneš/Waksman permutation network [2, 20]. This network consists of two back-to-back butterfly network of width 2^k, where each level in the network has 2^{k-1} "switch gates" and each switch gate swaps (or not) its two inputs, depending on a control bit. It is possible to realize any permutation of $\ell = 2^k$ items by appropriately setting the control bits of all the switch gates. Viewing this network as acting on k-bit addresses, the i-th level of the network partitions the 2^k addresses into 2^{k-1} pairs, where each pair of addresses differs only in the $|i-k|$-th bit, and then it swaps (or not) those pairs. The fact that the pairs in the i-th level always consist of addresses that differ by exactly $2^{|i-k|}$, makes it easy to implement each level using rotations: All we need is one rotation by $2^{|i-k|}$ and another by $-2^{|i-k|}$, followed by two batched Select operations.

For general rings $\mathbb{A}$, the automorphisms do not always exactly "rotate" the plaintext slots. Instead, they act on the slots in a way that depends on a quotient group $\mathcal{H}$ of the appropriate Galois group. Nonetheless, we use basic theorems from Galois theory, in conjunction with appropriate generalizations of the Beneš/Waksman procedure, to construct a permutation network of depth $O(\log \ell)$ that can realize any permutation over the ℓ plaintext slots, where each level of the network consists of a constant number of permutations from $\mathcal{H}$ and Select operations. As with the rotations considered in [3], applying permutations from $\mathcal{H}$ can be done in time quasi-linear in ciphertext size, which is only quasi-linear in the security parameter. Overall, we find that permutation networks and Galois theory are a surprisingly fruitful combination.

We note that Damgård, Ishai and Krøigaard [7] used permutation networks in a somewhat analogous fashion to perform secure multiparty computation with *packed secret shares*. In their setting, which permits interaction between the parties, the permutations can be evaluated using much simpler mathematical machinery.

1.3 FHE with Polylog Overhead

In our discussion above, we glossed over the fact that ciphertext sizes in a BGV-like cryptosystem [3] depend polynomially on the depth of the circuit being evaluated, because the modulus size must grow with the depth of the circuit (unless bootstrapping [8, 9] is used). So, without bootstrapping, the "polylog overhead" result only applies to circuits of polylog depth. However, decryption itself can be accomplished in log-depth [3], and moreover the parameters can be set so that a ciphertext with $\tilde{\Omega}(\lambda)$ slots can be decrypted using a circuit of size $\tilde{O}(\lambda)$. Therefore, "recryption" can be accomplished with polylog overhead, and we obtain FHE with polylog overhead for arbitrary (wide enough) circuits.

2 Computing on (Encrypted) Arrays

As we explained above, our main tool for low-overhead homomorphic computation is to compute on "packed ciphertexts", namely make each ciphertext hold a vector of plaintext values rather than a single value. Throughout this section we let ℓ be a parameter specifying the number of plaintext values that are packed inside each ciphertext, namely we always work with ℓ-vectors of plaintext values. Let $\mathbb{K}_n = \mathbb{F}_{p^n}$ denote the plaintext space (e.g., $\mathbb{K}_n = \mathbb{F}_2$ if we are dealing with binary circuits directly). It was shown in [3, 18] how to homomorphically evaluate batch addition and multiplication operations on ℓ-vectors:

$$\ell\text{-Add}\big(\langle u_0,\ldots,u_{\ell-1}\rangle, \langle v_0,\ldots,v_{\ell-1}\rangle\big) \stackrel{\text{def}}{=} \langle u_0+v_0,\ldots,u_{\ell-1}+v_{\ell-1}\rangle$$
$$\ell\text{-Mult}\big(\langle u_0,\ldots,u_{\ell-1}\rangle, \langle v_0,\ldots,v_{\ell-1}\rangle\big) \stackrel{\text{def}}{=} \langle u_0\times v_0,\ldots,u_{\ell-1}\times v_{\ell-1}\rangle$$

on packed ciphertexts in time $\tilde{O}((\ell+\lambda)(\log|\mathbb{K}_n|)$ where λ is the security parameter (with addition and multiplication in $\mathbb{K}_n$).[5] Specifically, if the size of our plaintext space is polynomially bounded and we set $\ell=\Theta(\lambda)$, then we can evaluate the above operations homomorphically in time $\tilde{O}(\lambda)$.

Unfortunately, component-wise ℓ-Add and ℓ-Mult are not sufficient to perform arbitrary computations on encrypted arrays, since data at different indexes within the arrays can never interact. To get a *complete set of operations for arrays*, we introduce the ℓ-Permute operation that can arbitrarily permute the data within the ℓ-element arrays. Namely, for any permutation π over the indexes $I_\ell=\{0,1,\ldots,\ell-1\}$, we want to homomorphically evaluate the function

$$\ell\text{-Permute}_\pi\big(\langle u_0,\ldots,u_{\ell-1}\rangle\big)=\big\langle u_{\pi(0)},\ldots,u_{\pi(\ell-1)}\big\rangle.$$

on a packed ciphertext, with complexity similar to the above. We will show how to implement ℓ-Permute homomorphically in Sections 3 and 4 below. For now, we just assume that such an implementation is available and show how to use it to obtain low-overhead implementation of general circuits.

2.1 Computing with ℓ-Fold Gates

We are interested in computing arbitrary functions using "ℓ-fold gates" that operate on ℓ-element arrays as above. We assume that the function $f(\cdot)$ to be computed is specified using a fan-in-2 arithmetic circuit with t "normal" arithmetic gates (that operate on singletons). Our goal is to implement f using as few ℓ-fold gates as possible, hopefully not much more than t/ℓ of them.

We assume that the input to f is presented in a packed form, namely when computing an r-variate function $f(x_1,\ldots,x_r)$ we get as input $\lceil r/\ell\rceil$ arrays (indexed $A_0,\ldots,A_{\lceil r/\ell\rceil}$) with the j'th array containing the input elements $x_{j\ell}$ through $x_{j\ell+\ell-1}$. The last array may contain less than ℓ elements, and the unused entries contain "don't care" elements. In fact, throughout the computation we allow all of the arrays to contain "don't care" entries. We say that an array is *sparse* if it contains $\ell/2$ or more "don't care" entries. We maintain the invariant that our collection of arrays is always at least half full, i.e., we hold r values using at most $\lceil 2r/\ell\rceil$ ℓ-element arrays.

The gates that we use in the computation are the ℓ-Add, ℓ-Mult, and ℓ-Permute gates from above. The rest of this section is devoted to establishing the following theorem:

Theorem 1. *Let ℓ,t,w and W be parameters. Then any t-gate fan-in-2 arithmetic circuit C with average width w and maximum width W, can be evaluated using a network of $O\big(\lceil t/\ell\rceil\cdot\lceil\ell/w\rceil\cdot\log W\cdot\mathrm{polylog}(\ell)\big)$ ℓ-fold gates of types ℓ-Add, ℓ-Mult, and ℓ-Permute. The depth of this network of ℓ-fold gates is at most $O(\log W)$ times that of the original circuit C, and the description of the network can be computed in time $\tilde{O}(t)$ given the description of C.*

Before turning to proving Theorem 1, we point out that Theorem 1 implies that if the original circuit C has size $t=\mathrm{poly}(\lambda)$, depth L, and average width $w=\Omega(\lambda)$, and if we set the packing parameter as $\ell=\Theta(\lambda)$, then we get an $O(L\cdot\log\lambda)$-depth implementation of C using $O(t/\lambda\cdot\mathrm{polylog}(\lambda))$ ℓ-fold gates. If implementing each ℓ-fold gate takes $\tilde{O}(L\lambda)$ time, then the total time to evaluate C is no more than

$$O\big(\frac{t}{\lambda}\mathrm{polylog}(\lambda)\cdot L\cdot\lambda\cdot\mathrm{polylog}(\lambda)\big)=O(t\cdot L\cdot\mathrm{polylog}(\lambda)).$$

Therefore, with this choice of parameter (and for "wide enough" circuits of average width $\Omega(\lambda)$), our overhead for evaluating depth-L circuits is only $O(L\cdot\mathrm{polylog}(\lambda))$. And if L is also polylogarithmic, as in BGV with bootstrapping [3], then the total overhead is polylogarithmic in the security parameter.

[5] To compute L levels of such operations, the complexity expression becomes $\tilde{O}((\ell+\lambda)(L+\log|\mathbb{K}_n|))$.

The high-level idea of the proof of Theorem 1 is what one would expect. Consider an arbitrary fan-in two arithmetic circuit C. Suppose that we have $\approx w$ output wire values of level $i-1$ packed into roughly w/ℓ arrays. We need to route these output values to their correct input positions at level i. It should be obvious that the ℓ-Permute gates facilitate this routing, except for two complications:

1. The mapping from outputs of level $i-1$ to inputs of level i is not a permutation. Specifically, level-$(i-1)$ gates may have high fan-out, and so some of the output values may need to be *cloned*.
2. Once the output values are cloned sufficiently (for a total of, say, w' values), routing to level i apparently calls for a *big permutation* over w' elements, not just a small permutation within arrays of ℓ elements.

Below we show that these complications can be handled efficiently.

2.2 Permutations over Hyper-Rectangles

First, consider the second complication from above – namely, that we need to perform a permutation over some w elements (possibly $w \gg \ell$) using ℓ-Add, ℓ-Mult, and ℓ-Permute operations that only work on ℓ-element arrays. We use the following basic fact (cf. [14]), for completeness we provide a proof in Appendix B.

Lemma 1. *Let $S = \{0,\ldots,a-1\} \times \{0,\ldots,b-1\}$ be a set of ab positions, arranged as a matrix of a rows and b columns. For any permutation π over S, there are permutations π_1, π_2, π_3 such that $\pi = \pi_3 \circ \pi_2 \circ \pi_1$ (that is, π is the composition of the three permutations) and such that π_1 and π_3 only permute positions within each column (these permutations only change the row, not the column, of each element) and π_2 only permutes positions within each row. Moreover, there is a polynomial-time algorithm that given π outputs the decomposition permutations π_1, π_2, π_3.*

In our context, Lemma 1 says that if we have w elements packed into $k = \lceil w/\ell \rceil$ ℓ-element arrays, we can express any permutation π of these elements as $\pi = \pi_3 \circ \pi_2 \circ \pi_1$ where π_2 invokes ℓ-Permute (k times in parallel) to permute data within the respective arrays, and π_1, π_3 only permute (ℓ times in parallel) elements that share the same index within their respective arrays. In Section 2.3, we describe how to implement π_1, π_3 using ℓ-Add and ℓ-Mult, and analyze the overall efficiency of implementing π. The following generalization of Lemma 1 to higher dimensions will be used later in this work. It is proved by invoking Lemma 1 recursively.

Lemma 2. *Let $S = I_{n_1} \times \cdots \times I_{n_k}$ where $I_{n_i} = \{0,\ldots,n_i-1\}$. (Each element in S has k coordinates.) For any permutation π over S, there are permutations $\pi_1,\ldots,\pi_{2k-1}$ such that $\pi = \pi_{2k-1} \circ \cdots \circ \pi_1$ and such that π_i affects only the i-th coordinate for $i \le k$ and only the $(2k-i)$-th coordinate for $i \ge k$.*

2.3 Batch Selections, Swaps, and Permutation Networks

We now describe how to use ℓ-Add and ℓ-Mult to realize the outer permutations π_1, π_3, which permute (ℓ times in parallel) elements that share the same index within their respective arrays. To perform these permutations, we can apply a *permutation network* à la Beneš/Waksman [2, 20]. Recall that a r-dimensional Beneš network consists of two back-to-back butterfly networks. Namely it is a $(2r-1)$-level network with 2^r nodes in each level, where for $i = 1, 2, \ldots, 2r-1$, we have an edge connecting node j in level $i-1$ to node j' in level i if the indexes j, j' are either equal (a "straight edge") or they differ in only in the $|r-i|$'th bit (a "cross edge"). The following lemma is an easy corollary of Lemma 2.

Lemma 3. [13, Thm 3.11] *Given any one-to-one mapping π of 2^r inputs to 2^r outputs in an r-dimensional Beneš network (one input per level-0 node and one output per level-$(2r-1)$ node), there is a set of node-disjoint paths from the inputs to the outputs connecting input i to output $\pi(i)$ for all i.*

In our setting, to implement our π_1 and π_3 from Lemma 1 we need to evaluate ℓ of these permutation networks in parallel, one for each index in our ℓ-fold arrays. Assume for simplicity that the number of ℓ-fold arrays is a power of two, say 2^r, and denote these arrays by $A_0, \ldots, A_{2^r-1}$, we would have a $(2r-1)$-level network, where the i'th level in the network consists of operating on pairs of arrays $(A_j, A_{j'})$, such that the indexes j, j' differ only in the $|r-i|$'th bit.

The operation applied to two such arrays $A_j, A_{j'}$ works separately on the different indexes of these arrays. For each $k = 0, 1, \ldots, \ell-1$ the operation will either swap $A_j[k] \leftrightarrow A_{j'}[k]$ or will leave these two entries unchanged, depending on whether the paths in the k'th permutation network uses the cross edges or the straight edges between nodes j and j' in levels $i-1, i$ of the permutation network.

Thus, evaluating ℓ such permutation networks in parallel reduces to the following Select function: Given two arrays $A = [m_0, \ldots, m_{\ell-1}]$ and $A' = [m'_0, \ldots, m'_{\ell-1}]$ and a string $S = s_0 \cdots s_{\ell-1} \in \{0,1\}^\ell$, the operation $\mathsf{Select}_S(A, A')$ outputs an array $A'' = [m''_0, \ldots, m''_{\ell-1}]$ where, for each k, $m''_k = m_k$ if $s_k = 1$ and $m''_k = m'_k$ otherwise. It is easy to implement $\mathsf{Select}_S(A, A')$ using just the ℓ-Add and ℓ-Mult operations – in particular

$$\mathsf{Select}_S(A, A') = \ell\text{-Add}\left(\ \ell\text{-Mult}(A, S),\ \ell\text{-Mult}(A', \bar{S})\ \right)$$

where $\bar{S}$ is the bitwise complement of S. Note that $\mathsf{Select}_{\bar{S}}(A, A')$ outputs precisely the elements that are discarded by $\mathsf{Select}_S(A, A')$. So, $\mathsf{Select}_S(A, A')$ and $\mathsf{Select}_{\bar{S}}(A, A')$ are exactly like the arrays A' and A', except that some pairs of elements with identical indexes have been *swapped* – namely, those pairs at index k where $S_k = 0$. Hence we obtain the following, again the proof is deferred to Appendix B.

Lemma 4. *Evaluating ℓ permutation networks in parallel, each permuting k items, can be accomplished using $O(k \cdot \log k)$ gates of ℓ-Add and ℓ-Mult, and depth $O(\log k)$. Also, evaluating a permutation π over $k \cdot \ell$ elements that are packed into k ℓ-element arrays, can be accomplished using k ℓ-Permute gates and $O(k \log k)$ gates of ℓ-Add and ℓ-Mult, in depth $O(\log k)$. Moreover, there is an efficient algorithm that given π computes the circuit of ℓ-Permute, ℓ-Add, and ℓ-Mult gates that evaluates it, specifically we can do it in time $O(k \cdot \ell \cdot \log(k \cdot \ell))$.*

2.4 Cloning: Handling High Fan-out in the Circuit

We have described how to efficiently realize a permutation over $w > \ell$ items using ℓ-Add, ℓ-Mult and ℓ-Permute gates that operate on ℓ-element arrays. However, the wiring between adjacent levels of a fan-in-two circuit are typically not permutations, since we typically have gates with high fan-out. We therefore need to clone the output values of these high-fan-out gates before performing a permutation that maps them to their input positions at the next level. We describe an efficient procedure for this "cloning" step.

A cloning procedure. The input to the cloning procedure consists of a collection of k arrays, each with ℓ slots, where each slot is either "full" (i.e., contains a value that we want to use) or "empty" (i.e., contains a don't-care value). We assume that initially more than $k \cdot \ell/2$ of the available slots are full, and will maintain a similar invariant throughout the procedure. Denote the number of full slots in the input arrays by w (with $k \cdot \ell/2 < w \leq k \cdot \ell$), and denote the i'th input value by v_i. The ordering of input values is arbitrary – e.g., we concatenate all the arrays and order input values by their index in the concatenated multi-array.

We are also given a set of positive integers $m_1, \ldots, m_w \geq 1$, such that v_1 should be duplicated m_1 times, v_2 should be duplicated m_2 times, etc. We say that m_i is the *intended multiplicity* of v_i. The total number of full slots

in the output arrays will therefore be $w' \stackrel{\text{def}}{=} m_1 + m_2 + \cdots + m_w \geq w$. In more detail, the output of the cloning procedure must consist of some number k' of ℓ-slot arrays, where $k'\ell/2 < w' \leq k'\ell$, such that v_1 appears in at least m_1 of the output slots, v_2 appears in at least m_2 of the output slots, etc.

Denote the largest intended multiplicity of any value by $M = \max_i\{m_i\}$. The cloning procedure works in $\lceil \log M \rceil$ phases, such that after the j'th phase each value v_i is duplicated $\min(m_i, 2^j)$ times. Each phase consists of making a copy of all the arrays, then for values that occur too many times marking the excess slots as empty (i.e., marking the extra occurrences as don't-care values), and finally merging arrays that are "sparse" until the remaining arrays are at least half full. A simple way to merge two sparse arrays is to permute them so that the full slots appear in the left half in one array and the right half in the other, and then apply Select in the obvious way. A pseudo-code description of this procedure is given in Figure 1, whilst the proof of the following lemma is in Appendix B.

Input: k ℓ-slot arrays, $A_1, \ldots, A_k$, each of the $k \cdot \ell$ slots containing either a value or the special symbol '$\perp$', w positive integers $m_1, \ldots, m_w \geq 1$, where w is the number of full slots in the input arrays.
Output: k' ℓ-slot arrays, $A'_1, \ldots, A'_{k'}$, with each slot containing either a value or the special symbol '$\perp$', where $k'/2 \leq (\sum_i m_i)/\ell \leq k'$ and each input value v_i is replicated m_i times in the output arrays

0. Set $M \leftarrow \max_i\{m_i\}$
1. For $j = 1$ to $\lceil \log M \rceil$ // The j'th phase
2. Make another copy of all the arrays // Duplicate everything
3. While there are values v_i with multiplicity more than m_i:
4. Replace the excess occurrences of v_i by $\perp$ // Remove redundant entries
5. While there exist pairs of arrays that have between them ℓ or more slots with $\perp$:
6. Pick one such pair and merge the two arrays //Merge sparse arrays
7. Output the remaining arrays

Fig. 1. The cloning procedure

Lemma 5. ***(i)*** *The cloning procedure from Figure 1 is correct.*

(ii) *Assuming that at least half the slots in the input arrays are full, this procedure can be implemented by a network of $O(w'/\ell \cdot \log(w'))$ ℓ-fold gates of type ℓ-Add, ℓ-Mult and ℓ-Permute, where w' is the total number of full slots in the output, $w' = \sum m_i$. The depth of the network is bounded by $O(\log w')$.*

(iii) *This network can be constructed in time $\tilde{O}(w')$, given the input arrays and the m_i's.*

We also describe some more optimizations in Appendix A, including a different cloning procedure that improves on the complexity bound in Lemma 5. Putting all the above together we can efficiently evaluate a circuit using ℓ-Permute, ℓ-Add and ℓ-Mult, yielding a proof of Theorem 1, see Appendix B.

3 Permutation Networks from Abelian Group Actions

As we will show in Section 4, the algebra underlying our FHE scheme makes it possible to perform inexpensive operations on packed ciphertexts, that have the effect of permuting the ℓ plaintext slots inside this packed ciphertext. However, not every permutation can be realized this way; the algebra only gives us a small set of "simple" permutations. For example, in some cases, the given automorphisms "rotate" the plaintext slots, transforming a ciphertext that encrypts the vector $\langle v_0, \ldots, v_{\ell-1} \rangle$ into one that encrypts $\langle v_k, \ldots, v_{\ell-1}, v_0, \ldots, v_{k-1} \rangle$, for any value of k of our choosing. (See Section 3.2 for the general case.)

Our goal in this section is therefore to efficiently implement an ℓ-Permute$_\pi$ operation for an arbitrary permutation π using only the simple permutations that the algebra gives us (and also the ℓ-Add and ℓ-Mult operations that we have available). We begin in Section 3.1 by showing how to efficiently realize arbitrary permutations when the small set of "simple permutations" is the set of rotations. In Section 3.2 we generalize this construction to a more general set of simple permutations.

3.1 Permutation Networks from Cyclic Rotations and Swaps

Consider the Beneš permutation network discussed in Lemma 3. It has the interesting property that when the 2^r items being permuted are labeled with r-bit strings, then the i-th level only swaps (or not) pairs whose index differs in the $|r-i|$-th bit. In other words, the i-th level swaps only disjoint pairs that have offset $2^{|r-i|}$ from each other. We call this operation an "offset-swap", since all pairs of elements that might be swapped have the same mutual offset.

Definition 1 (Offset Swap). *Let $I_\ell = \{0,\ldots,\ell-1\}$. We say that a permutation π over I_ℓ is an i-offset swap if it consists only of 1-cycles and 2-cycles (i.e., $\pi = \pi^{-1}$), and moreover all the 2-cycles in π are of the form $(k, k+i \bmod \ell)$ for different values $k \in I_\ell$.*

Offset swaps modulo ℓ are easy to implement by combining two rotations with the Select operation defined in Section 2.3. Specifically, for an i-offset swap, we need rotations by i and $-i \bmod \ell$ and two Select operations. By Lemma 3, a Beneš network can realize any permutation over 2^r elements using $2r-1$ levels where the i-th level is a $2^{|k-i|}$-offset swap modulo 2^r. An i-offset modulo 2^r, $\ell < 2^r < 2\ell$ can be cobbled together using a constant number of offset swaps modulo ℓ and Select operations, with offsets i and $2\ell - i$. Therefore, given a cyclic group of "simple" permutations $\mathcal{H}$ and Select operations, we can implement any permutation using a Beneš network with low overhead. Specifically, we prove the following lemma in Appendix B.

Lemma 6. *Fix an integer ℓ and let $k = \lceil \log \ell \rceil$. Any permutation π over $I_\ell = \{0,\ldots,\ell-1\}$ can be implemented by a $(2k-1)$-level network, with each level consisting of a constant number of rotations and* Select *operations on ℓ-arrays.*

Moreover, regardless of the permutation π, the rotations that are used in level i ($i = 1,\ldots,2k-1$) are always exactly $2^{|k-i|}$ and $\ell - 2^{|k-i|}$ positions, and the network depends on π only via the bits that control the Select *operations. Finally, this network can be constructed in time $\tilde{O}(\ell)$ given the description of π.*

3.2 Generalizing to Sharply-Transitive Abelian Groups

Below, we extend our techniques above to deal with a more general set of "simple permutations" that we get from our ring automorphisms. (See Sections 4 and C.3.)

Definition 2 (Sharply Transitive Permutation Groups). *Denote the ℓ-element symmetric group by $\mathcal{S}_\ell$ (i.e., the group of all permutations over $I_\ell = \{0,\ldots,\ell-1\}$), and let $\mathcal{H}$ be a subgroup of $\mathcal{S}_\ell$. The subgroup $\mathcal{H}$ is* sharply transitive *if for every two indexes $i, j \in I_\ell$ there exists a unique permutation $h \in \mathcal{H}$ such that $h(i) = j$.*

Of course, the group of rotations is an example of an abelian and sharply transitive permutation group. It is abelian: rotating by k_1 positions and then by k_2 positions is the same as rotating by k_2 positions and then by k_1 positions. It is also sharply transitive: for all i, j there is a single rotation amount that maps index i to index j,

namely rotation by $j-i$. However, rotations are certainly not the only example. We now explain how to efficiently realize arbitrary permutations using as building blocks the permutations from any sharply-transitive abelian group.

Recall that any abelian group is isomorphic to a direct product of cyclic groups, hence $\mathcal{H} \cong C_{\ell_1} \times \cdots \times C_{\ell_k}$ (where C_{ℓ_i} is a cyclic group with ℓ_i elements for some integers $\ell_i \geq 2$ where ℓ_i divides ℓ_{i+1} for all i). As any cyclic group with ℓ_i elements is isomorphic to $I_{\ell_i} = \{0, 1, \ldots, \ell_i - 1\}$ with the operation of addition mod ℓ_i, we will identify elements in $\mathcal{H}$ with vectors in the box $\mathcal{B} = I_{\ell_1} \times \cdots \times I_{\ell_k}$, where composing two group elements corresponds to adding their associated vectors (modulo the box). The group $\mathcal{H}$ is generated by the k unit vectors $\{\boldsymbol{e}_r\}_{r=1}^k$ (where $\boldsymbol{e}_r = \langle 0, \ldots, 0, 1, 0, \ldots, 0\rangle$ with 1 in the r-th position). We stress that our group $\mathcal{H}$ has polynomial size, so we can efficiently compute the representation of elements in $\mathcal{H}$ as vectors in $\mathcal{B}$.

Since $\mathcal{H}$ is a sharply transitive group of permutations over the indexes $I_\ell = \{0, \ldots, \ell - 1\}$, we can similarly label the indexes in I_ℓ by vectors in $\mathcal{B}$: Pick an arbitrary index $i_0 \in I_\ell$, then for all $h \in \mathcal{H}$ label the index $h(i_0) \in I_\ell$ with the vector associated with h. This procedure labels every element in I_ℓ with exactly one vector from $\mathcal{B}$, since for every $i \in I_\ell$ there is a unique $h \in \mathcal{H}$ such that $h(i_0) = i$. Also, since $\mathcal{H} \cong \mathcal{B}$, we use all the vectors in $\mathcal{B}$ for this labeling ($|\mathcal{H}| = |\mathcal{B}| = \ell$). Note that with this labeling, applying the generator $\boldsymbol{e}_r$ to an index labeled with vector $\boldsymbol{v} \in \mathcal{B}$, yields an index labeled with $\boldsymbol{v'} = \boldsymbol{v} + \boldsymbol{e}_r \bmod \mathcal{B}$. Namely we increment by one the r'th entry in $\boldsymbol{v}$ $(\bmod\ \ell_r)$, leaving the other entries unchanged.

In other words, rather than a one-dimensional array, we view I_ℓ as a k-dimensional matrix (by identifying it with $\mathcal{B}$). The action of the generator $\boldsymbol{e}_r$ on this matrix is to rotate it by one along the r-th dimension, and similarly applying the permutation $\boldsymbol{e}_r^k \in \mathcal{H}$ to this matrix rotates it by k positions along the r-th dimension. For example, when $k = 2$, we view I_ℓ as an $\ell_1 \times \ell_2$ matrix, and the group $\mathcal{H}$ includes permutations of the form $\boldsymbol{e}_1^k$ that rotate all the columns of this matrix by k positions and also permutations of the form $\boldsymbol{e}_2^k$ that rotate all the rows of this matrix by k positions.

Using Lemma 6, we can now implement arbitrary permutations along the r'th dimension using a permutation network built from offset-swaps along the r'th dimension. Moreover, since the offset amounts used in the network do not depend on the specific permutation that we want to implement, we can use just one such network to implement in parallel different arbitrary permutations on different r'th-dimension sub-matrices. For example, in the 2-dimensional case, we can effect a different permutation on every column, yet realize all these different permutations using just one network of rotations and Selects, by using the same offset amounts but different Select bits for the different columns. More generally we can realize arbitrary (different) ℓ/ℓ_r permutations along all the different "generalized columns" in dimension-r, using a network of depth $O(\log \ell_r)$ consisting of permutations $h \in \mathcal{H}$ and ℓ-fold Select operations (and we can construct that network in time $\ell/\ell_r \cdot \tilde{O}(\ell_r) = \tilde{O}(\ell)$).

Once we are able to realize different arbitrary permutations along the different "generalized columns" in all the dimensions, we can apply Lemma 2. That lemma allows us to decompose any permutation π on I_ℓ into $2k-1$ permutations $\pi = \pi_i \circ \cdots \circ \pi_{2k-1}$ where each π_i consists only of permuting the generalized columns in dimension $r = |k - i|$. Hence we can realize an arbitrary permutation on I_ℓ as a network of permutations $h \in \mathcal{H}$ and ℓ-fold Select operations, of total depth bounded by $2\sum_{i=0}^{k-1} O(\log \ell_i) = O(\log \ell)$ (the last bound follows since $\ell = \prod_{i=0}^{k-1} \ell_i$). Also we can construct that network in time bounded by $2\sum_{i=0}^{k-1} \tilde{O}(\ell_i) = \tilde{O}(\ell)$ (the bound follows since $k \leq \log \ell$). Concluding this discussion, we have:

Lemma 7. *Fix any integer ℓ and any abelian sharply-transitive group of permutations over I_ℓ, $\mathcal{H} \subset \mathcal{S}_\ell$. Then for every permutation $\pi \in \mathcal{S}_\ell$, there is a permutation network of depth $O(\log \ell)$ that realizes π, where each level of the network consists of a constant number of permutations from $\mathcal{H}$ and* Select *operations on ℓ-arrays.*

Moreover, the permutations used in each level do not depend on the particular permutation π, the network depends on π only via the bits that control the Select *operations. Finally, this network can be constructed in time $\tilde{O}(\ell)$ given the description of π and the labeling of elements in $\mathcal{H}, I_\ell$ as vectors in $\mathcal{B}$.* □

Lemma 7 tells us that we can implement an arbitrary ℓ-Permute operation using a log-depth network of permutations $h \in \mathcal{H}$ (in conjunction with ℓ-Add and ℓ-Mult). Plugging this into Theorem 1 we therefore obtain:

Theorem 2. *Let ℓ, t, w and W be parameters, and let $\mathcal{H}$ be an abelian, sharply-transitive group of permutations over I_ℓ.*

Then any t-gate fan-in-2 arithmetic circuit C with average width w and maximum width W, can be evaluated using a network of $O\big(\lceil t/\ell \rceil \cdot \lceil \ell/w \rceil \cdot \log W \cdot \mathrm{polylog}(\ell)\big)$ ℓ-fold gates of types ℓ-Add, ℓ-Mult, and $h \in \mathcal{H}$. The depth of this network of ℓ-fold gates is at most $O(\log W \cdot \log \ell)$ times that of the original circuit C, and the description of the network can be computed in time $\tilde{O}(t \cdot \log \ell)$ given the description of C. □

4 FHE With Polylog Overhead

Theorem 2 implies that if we could efficiently realize ℓ-Add, ℓ-Mult, and $\mathcal{H}$-actions on packed ciphertexts (where $\mathcal{H}$ is a sharply transitive abelian group of permutations on ℓ-slot arrays), then we can evaluate arbitrary (wide enough) circuits with low overhead. Specifically, if we could set $\ell = \Theta(\lambda)$ and realize ℓ-Add, ℓ-Mult, and $\mathcal{H}$-actions in time $\tilde{O}(\lambda)$, then we can realize any circuit of average width $\Omega(\lambda)$ with just $\mathrm{polylog}(\lambda)$ overhead. It remains only to describe an FHE system that has the required complexity for these basic homomorphic operations.

4.1 The Basic Setting of FHE Schemes Based on Ideal Lattices and Ring LWE

Many of the known FHE schemes work over a polynomial ring $\mathbb{A} = \mathbb{Z}[X]/F(X)$, where $F(X)$ is irreducible monic polynomial, typically a cyclotomic polynomial. Ciphertexts are typically vectors (consisting of one or two elements) over $\mathbb{A}_q = \mathbb{A}/q\mathbb{A}$ where q is an integer modulus, and the plaintext space of the scheme is $\mathbb{A}_p = \mathbb{A}/p\mathbb{A}$ for some integer modulus $p \ll q$ with $\gcd(p, q) = 1$, for example $p = 2$. (Namely, the plaintext is represented as an integer polynomial with coefficients mod p.) Secret keys are also vectors over $\mathbb{A}_q$, and decryption works by taking the inner product $b \leftarrow \langle \mathbf{c}, \mathbf{s} \rangle$ in $\mathbb{A}_q$ (so b is an integer polynomial with coefficients in $(-q/2, q/2]$) then recovering the message as $b \bmod p$. Namely, the decryption formula is $[[\langle \mathbf{c}, \mathbf{s} \rangle \bmod F(X)]_q]_p$ where $[\cdot]_q$ denotes modular reduction into the range $(-q/2, q/2]$. Below we consider ciphertext vectors and secret-key vectors with two entries, since this is indeed the case for the variant of the BGV scheme [3] that we use.

Smart and Vercauteren [18] observed that the underlying ring structure of these schemes makes it possible to realize homomorphic (batch) Add and Mult operations, i.e. our ℓ-Add and ℓ-Mult. Specifically, though $F(X)$ is typically irreducible over $\mathbb{Q}$, it may nonetheless factor modulo p; $F(X) = \prod_{i=0}^{\ell-1} F_i(X) \bmod p$. In this case, the plaintext space of the scheme also factors: $\mathbb{A}_p = \otimes_{j=0}^{\ell-1} \mathbb{A}_{\mathfrak{p}_j}$ where $\mathfrak{p}_i$ is the ideal in $\mathbb{A}$ generated by p and $F_i(X)$. In particular, the Chinese Remainder Theorem applies, and the plaintext space is partitioned into ℓ independent non-interacting "plaintext slots", which is precisely what we need for component-wise ℓ-Add and ℓ-Mult. The decryption formula recovers the "aggregate plaintext" $a \leftarrow [[\langle \mathbf{c}, \mathbf{s} \rangle \bmod F(X)]_q]_p$, and this aggregate plaintext is decoded to get the individual plaintext elements, roughly via $z_j \leftarrow a \bmod (F_i(x), p) \in \mathbb{A}_{\mathfrak{p}_j}$.

4.2 Implementing Group Actions on FHE Plaintext Slots

While component-wise Add and Mult are straightforward, getting different plaintext slots to interact is more challenging. For ease of exposition, suppose at first that $F(X)$ is the degree-$(m-1)$ polynomial $\Phi_m(X) = (X^m - 1)/(X - 1)$ for m prime, and that $p \equiv 1 \pmod{m}$. Thus our ring $\mathbb{A}$ above is the mth cyclotomic number field. In this case $F(X)$ factors to linear terms modulo p, $F(X) = \prod_{i=0}^{\ell-1}(X - \rho_i) \pmod{p}$ with $\rho_i \in \mathbb{F}_p$. Hence

we obtain $\ell = m - 1$ plaintext slots, each slot holding an element of the finite field $\mathbb{F}_p$ (i.e. in this case $\mathbb{A}_{\mathfrak{p}_i}$ above is equal to $\mathbb{F}_p$).

To get Φ_m to factor modulo p into linear terms we must have $p \equiv 1 \pmod{m}$, so $p > m$. Also we need $m = \Omega(\lambda)$ to get security (since m is roughly the dimension of the underlying lattice). This means that to get Φ_m to factor into linear terms we must use plaintext spaces that are somewhat large (in particular we cannot directly use $\mathbb{F}_2$). Later in this section we sketch the more elaborate algebra needed to handle the general (and practical) case of non-prime m and $p \ll m$, where Φ_m may not factor into linear terms. This is covered in more detail in Appendix C. For now, however, we concentrate on the simple case where Φ_m factors into linear terms modulo p.

Recall that ciphertexts are vectors over $\mathbb{Z}_q[X]/\Phi_m(X)$, so each entry in these vectors corresponds to an integer polynomial. Consider now what happens if we simply replace X with X^i inside all these polynomials, for some exponent $i \in Z_m^*, i > 1$. Namely, for each polynomial $f(X)$, we consider $f^{(i)}(X) = f(X^i) \bmod \Phi_m(X)$. Notice that if we were using polynomial arithmetic modulo $X^m - 1$ (rather then modulo $\Phi_m(X)$) then this transformation would just permutes the coefficients of the polynomials. Namely $f^{(i)}$ has the same coefficients as f but in a different order, which means that if the coefficient vector of f has small norm then the same holds for the coefficient vector of $f^{(i)}$. In Appendix D we show that using a different notion of "size" of a polynomial (namely, the norm of the canonical embedding of a polynomial rather than the norm of its coefficient vector), we can conclude the same also for mod-Φ_m polynomial arithmetic. Namely, the mapping $f(X) \mapsto f(X^i) \bmod \Phi_m(X)$ does not change the "size" of the polynomial. To simplify presentation, below we describe everything in terms of coefficient vectors and arithmetic modulo $X^m - 1$. The actual mod-Φ_m implementation that we use is described in Appendix D (and a slightly different implementation is described in Appendix E).

Let us now consider the effect of the transformation $X \mapsto X^i$ on decryption. Let $\mathbf{c} = (c_0(X), c_1(X))$ and $\mathbf{s} = (s_0(X), s_1(X))$ be ciphertext and secret-key vectors, and let $b = \langle \mathbf{c}, \mathbf{s} \rangle \bmod (X^m - 1, q)$ and $a = b \bmod p$. Denote $\mathbf{c}^{(i)} = (c_0(X^i), c_1(X^i)) \bmod (X^m - 1)$, and define $\mathbf{s}^{(i)}$, $b^{(i)}$ and $a^{(i)}$ similarly. Since $\langle \mathbf{c}, \mathbf{s} \rangle = b \pmod{X^m - 1, q}$, we have that

$$c_0(X)s_0(X) + c_1(X)s_1(X) \;=\; b(X) + q \cdot r(X) + (X^m - 1)s(X) \;\; (\text{over } \mathbb{Z}[X])$$

for some integer polynomials $r(X), s(X)$, and therefore also

$$c_0(X^i)s_0(X^i) + c_1(X^i)s_1(X^i) \;=\; b(X^i) + q \cdot r(X^i) + (X^{mi} - 1)s(X^i) \;\; (\text{over } \mathbb{Z}[X]).$$

Since $X^m - 1$ divides $X^{mi} - 1$, then we also have

$$\left\langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \right\rangle \;=\; b^{(i)} + q \cdot r(X^i) + (X^m - 1)S(X) \;\; (\text{over } \mathbb{Z}[X])$$

for some $r(X), S(X)$. That is, $b^{(i)} = \left\langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \right\rangle \bmod (X^m - 1, q)$. Clearly, we also have $a^{(i)} = b^{(i)} \pmod{p}$. This means that if $\mathbf{c}$ decrypts to the aggregate plaintext a under $\mathbf{s}$, then $\mathbf{c}^{(i)}$ decrypts to $a^{(i)}$ under $\mathbf{s}^{(i)}$!

The cryptosystem from [3, 4] have a mechanism for "key switching" (which is also applicable to the scheme from [5]), transforming a ciphertext $\mathbf{c}$ that decrypts to a under $\mathbf{s}$ to a new ciphertext $\mathbf{c}'$ that decrypts to the same a under some other secret key $\mathbf{s}'$. Using the same mechanism, we can translate the transformed ciphertext $\mathbf{c}^{(i)}$ into one that decrypts to $a^{(i)}$ under another $\mathbf{s}'$ of our choice. We can even translate it back to a ciphertext decryptable under the original $\mathbf{s}$ is we are willing to assume circular security. Using the BGV cryptosystem [5, 4, 3] with appropriate parameters, key switching can be accomplished in time $\tilde{O}(\lambda)$. (See Appendices D and E for details on our variants of the BGV scheme [5].)

But how does this new aggregate plaintext $a^{(i)}$ relate to the original a? Here we apply to Galois theory, which tells us that decoding the aggregate $a^{(i)}$ (which we do roughly by setting $z_j \leftarrow a^{(i)} \bmod (F_j, p)$), the set of z_j's

that we get is exactly the same as when decoding the original aggregate a, albeit in different order. Roughly, this is because each of our plaintext slots corresponds to a root of the polynomial $F(X)$, and the transformations $X \mapsto X^i$, which are precisely the elements of the Galois group, permute these roots. In other words by transforming $\mathbf{c} \to \mathbf{c}^{(i)}$ (followed by key switching), we can permute the plaintext slots inside the packed ciphertext. Moreover, in our simplified case, the permutations have a single cycle – i.e., they are rotations of the slots. Arranging the slots appropriately we can get that the transformation $\mathbf{c} \to \mathbf{c}^{(i)}$ rotates the slots by exactly i positions, thus we get the group of rotations that we were using in Section 3.1. In general the situation is a little more complicated, but the above intuition still can be made to hold; for more details see Appendix C.

The general case. In the general case, when m is not a prime, the polynomial $\Phi_m(X)$ has degree $\phi(m)$ (where $\phi(\cdot)$ is Euler's totient function), and it factors mod p into a number of same-degree irreducible factors. Specifically, the degree of the factors is the smallest integer d such that $p^d = 1 \pmod m$, and the number of factors is $\ell = \phi(m)/d$ (which is of course an integer), $\Phi_m(X) = \prod_{j=0}^{\ell-1} F_j(X)$. For us, it means that we have ℓ plaintext slots, each isomorphic to the finite field $\mathbb{F}_{p^d}$, and an aggregate plaintext is a degree-$(\phi(m)-1)$ polynomial over $\mathbb{F}_p$.

Suppose that we want to evaluate homomorphically a circuit over some underlying field $\mathbb{K}_n = \mathbb{F}_{p^n}$, then we need to find an integer m such that $\Phi_m(X)$ factors mod p into degree-d factors, where d is divisible by n. This way we could directly embed elements of the underlying plaintext space $\mathbb{K}_n$ inside our plaintext slots that hold elements of $\mathbb{F}_{p^d}$, and addition and multiplication of plaintext slots will directly correspond to additions and multiplications of elements in $\mathbb{K}_n$. (This follows since $\mathbb{K}_n = \mathbb{F}_{p^n}$ is a subfield of $\mathbb{F}_{p^d}$ when n divides d.)

Note that each plaintext slot will only have $n \log p$ bits of relevant information, i.e., the underlying element of $\mathbb{F}_{p^n}$, but it takes $d \log p$ bits to specify. We thus get an "embedding overhead" factor of d/n even before we encrypt anything. We therefore need to choose our parameter m so as to keep this overhead to a minimum.

Even for a non-prime m, the Galois group $\mathcal{G}\mathsf{al}(\mathbb{Q}[X]/\Phi_m(X))$ consists of all the transformations $X \mapsto X^i$ for $i \in \mathbb{Z}_m^*$, hence there are exactly $\phi(m)$ of them. As in the simplified case above, if we have a ciphertext $\mathbf{c}$ that decrypts to an aggregate plaintext a under $\mathbf{s}$, then $\mathbf{c}^{(i)}$ decrypts to $a^{(i)}$ under $\mathbf{s}^{(i)}$. Differently from the simple case, however, not all members of the Galois group induce permutations on the plaintext slots, i.e., decoding the aggregate plaintext $a^{(i)}$ does not necessarily give us the same set of (permuted) plaintext elements as decoding the original a. Instead $\mathcal{G}\mathsf{al}(\mathbb{Q}[X]/\Phi_m(X))$ contains a subgroup $\mathcal{G} = \{(X \mapsto X^{p^j}) : j = 0, 1, \ldots, d-1\}$ corresponding to the Frobenius automorphisms[6] modulo p. This subgroup does not permute the slots at all, but the quotient group $\mathcal{H} = \mathcal{G}\mathsf{al}/\mathcal{G}$ does. Clearly, $\mathcal{G}$ has order d and $\mathcal{H}$ has order $\phi(m)/d = \ell$. In Appendix C we show that the quotient group $\mathcal{H}$ acts as a transitive permutation group on our ℓ plaintext slots, and since it has order ℓ then it must be sharply transitive. In the general case we therefore use this group $\mathcal{H}$ as our permutation group for the purpose of Lemma 7. Another complication is that the automorphism that we can compute are elements of $\mathcal{G}\mathsf{al}$ and not elements in the quotient group $\mathcal{H}$. In Appendix C we also show how to emulate the permutations in $\mathcal{H}$, via use of coset representatives in $\mathcal{G}\mathsf{al}$.

4.3 Parameter Setting for Low-Overhead FHE

Given the background from above (and the modification of the BGV cryptosystem [5] in Appendices D or E), we explain how to set the parameters for our variant of the BGV scheme so as to get low-overhead FHE scheme. Below we first show how to evaluate depth-L circuits with average-width $\Omega(\lambda)$ with overhead of only $\tilde{O}(L)\cdot\text{polylog}(\lambda)$, and then use bootstrapping to get overhead of $\text{polylog}(\lambda)$ regardless of depth.

Plaintext-Space Terminology and Notations The discussion below refers to three different "plaintext spaces":

[6] The group G is called the *decomposition group* at p in the literature.

– The *"underlying plaintext space"*: The circuit that we want to evaluate homomorphically is an arithmetic circuit over some (finite) ring, and that finite ring is the "underlying plaintext space". We typically think of the underlying plaintext space as being just $\mathbb{F}_2$, but it is sometimes convenient to use other spaces (e.g., $\mathbb{F}_{2^8}$ when computing AES, or perhaps $\mathbb{F}_p$ for some 32-bit prime p in other applications).
  In this work we always assume that the underlying plaintext space is small, either of constant size or at most of size polynomial in λ. Moreover, we assume that it is a field, namely $\mathbb{K}_n = \mathbb{F}_{p^n}$ for some prime p and integer $n \geq 1$.
– The *"embedded plaintext space"*. This is what is held in each of our plaintext slots. For example, we could have underlying space $\mathbb{F}_2$, but embed our bits in elements of $\mathbb{F}_p$ for some larger integer p, or maybe in elements of $\mathbb{F}_{2^d}$ for some $d > 1$. (In the former case we need to emulate binary XOR using a degree-2 polynomial mod p, in the latter case multiplication and addition work as expected.)
– The *"aggregate plaintext space"*. This is the plaintext space that is natively encrypted in the cryptosystem: An element in the aggregate plaintext space is a polynomial in some $\mathbb{F}_p[X]$, and as explained above it encodes (via CRT) an ℓ-vector over the embedded plaintext space.

When choosing parameters for our FHE construction, we are given the depth and width of the circuits that we need to evaluate homomorphically, as well as the underlying plaintext space and the security parameter. We then want to choose the "embedded" and "aggregate" plaintext spaces and all the other parameters so as to minimize the overhead. Namely, minimize the ratio between the number of gates in the underlying circuits and the time that it takes to evaluate them homomorphically. We describe two methods for choosing the parameters: One is likely to be more efficient in practice, but we can only prove that it yields low overhead for either small underlying plaintext spaces (of size polylog(λ)) or very wide circuits (of width $\Omega(\lambda \cdot p^n)$). The other (simpler) method can be shown to work for any poly-size underlying plaintext space and circuits of width $\Omega(\lambda)$, but is almost certain to yield worst performance in practice.

In either approach, we begin by lower-bounding the dimension of the lattice that we need (in order to get security), thus getting a lower-bound on our parameter m (recall that we will eventually get a dimension-$\phi(m)$ lattice). Once we have this lower-bound M, we either pick $m = p^{ns} - 1 \geq M$ for some integer s, or just choose m as $p' - 1$ for some prime number p' sufficiently larger than M. In the former case we have "embedded plaintext space" $\mathbb{F}_{p^{ns}}$ into which we can directly embed the underlying space $\mathbb{F}_{p^n}$, and in the latter case we need to emulate $\mathbb{F}_{p^n}$ arithmetic using polynomials over $\mathbb{F}_{p'}$.

Once we set the parameter m and get the corresponding "embedded plaintext space", we can easily compute the packing parameter ℓ and all the other parameters.

Step 1. Lower-Bounding the Dimension Suppose that we want to evaluate homomorphically circuits of depth L over some small finite field $\mathbb{F}_{p^n}$, with average depth w and maximum depth $W = \text{poly}(\lambda)$, where λ is the security parameter. Clearly, for security parameter λ we need ciphertexts of size at least $\Omega(\lambda)$, so we cannot hope to evaluate any homomorphic operation faster than $\tilde{O}(\lambda)$. To get low overhead, we therefore must be able to pack at least $\ell = \tilde{\Omega}(\lambda)$ plaintext slots (from our "embedded" space) into one ciphertext. This means that we only get low-overhead implementation when the width of the underlying circuits is at least $\tilde{\Omega}(\lambda)$.

From Theorem 2 we know that for any packing parameter ℓ we can evaluate depth-L circuits using a network of ℓ-fold gates of depth $L' = O(L \cdot \log W \cdot \log \ell)$. (If we use the second approach below for choosing the parameter m then we need another additive term of $L \cdot \log(p^n) = O(L \cdot \log \lambda)$ to emulate $\mathbb{F}_{p^n}$ arithmetic using mod-m polynomials.) We will show below that it is sufficient to choose either $\ell = \Theta(\lambda)$ or $\ell = \Theta(p^n \cdot \lambda) \leq \text{poly}(\lambda)$ (depending on which of the two approaches we use), but in either case we have $L' \leq c \cdot L \cdot \log W \cdot \log \lambda$ for some constant c that we can compute from the given parameters.

Recall that the BGV cryptosystem needs L' different moduli q_i when evaluating a depth-L' network. When implementing arithmetic operations over a characteristic-p field and working with dimension-M lattices, the largest modulus needs to be $q_0 = (M \cdot p)^{c' \cdot L'}$ (for some constant $c' < 2$) to get the homomorphic evaluation functionality, and $M \geq \lambda \cdot \log q_0$ to get security. Plugging in all these constraints, we get a lower-bound on the dimension of the lattice $M \geq c'' \cdot L \cdot \lambda \log \lambda \cdot \log W \cdot \log p$ for some constant c'' that we can compute from the given parameters (note that $M = \tilde{\Theta}(L \cdot \lambda)$).

Step 2. Choosing the parameter m Below we will choose our parameter m so as to get $\phi(m) \geq M$. We use the following lemma, whose proof is in Appendix B.

Lemma 8. *For all positive integers m we have $m/\phi(m) = O(\log \log m)$.*

We will then choose our parameter m larger than $c^* M$ for some $c^* = O(\log \log M)$, to ensure that $\phi(m) \geq M$.

Approach 1: Using Extension Fields. Setting $s = \lceil \log_{p^n}(c^* M + 1) \rceil$, we see that the integer $m = p^{ns} - 1$ satisfies all our requirements. On one hand it is large enough, $m \geq c^* M$ by construction. On the other hand for $d = n \cdot s$ we clearly have that $p^d = 1 \pmod{m}$, which is what we need in order to use the "embedded plaintext space" $\mathbb{F}_{p^d}$ with the "aggregate plaintext space" $\mathbb{F}_p[X]/\Phi_m(X)$.

Moreover, the "embedding overhead" $d/n = s$ is small: since $M = \tilde{O}(L \cdot \lambda)$ and $s \leq \log_2(c^* M + 1)$ then clearly $s = O(\log(L \cdot \lambda))$. Thus the number of bits that it takes to specify an "aggregate plaintext" is only a factor of $O(\log(L \cdot \lambda))$ larger than what you need to specify all the elements of the "underlying plaintext space" that are embedded in this aggregate plaintext.

However, in some cases the parameter m itself (and therefore the lattice dimension) could be large: Note that we have $M = \tilde{O}(L \cdot \lambda)$ and since $s = \lceil \log_{p^n}(c^* M + 1) \rceil$ then $p^{ns} < (c^* M + 1) \cdot p^n$. If the size of the underlying plaintext space (i.e., p^n) is polylogarithmic, then we have $m = \tilde{O}(L \cdot \lambda)$ which is what we need. However, if the underlying plaintext size is larger, say $p^n \approx \lambda$, then we could have $m = \tilde{\Theta}(L \cdot \lambda^2)$. In this case we can no longer hope to evaluate homomorphic operations in time $\tilde{O}(L \cdot \lambda)$ (since the ciphertext size is too large).

If the circuits that we want to evaluate are very wide (i.e., of width $\tilde{\Omega}(\lambda \cdot p^n)$) then we can just pack sufficiently many plaintext slots inside each ciphertext to get the overhead down. We can do this since the "embedding overhead" is logarithmic. But for narrower circuits, say of width $\Theta(\lambda + p^n)$, we just don't have enough plaintext to put in all these slots, hence our overhead increases.

We point out that we may be able to do better than $m = p^{ns} - 1$, for example we can use any m' such that $\phi(m') > M$ and m' divides $p^{ns} - 1$. But it is not clear that such $m' < m$ exists (for example when $p = 2$ then $p^{ns} - 1$ could be a prime number). It is also permissible to choose some $s' > s$ and then choose m' that divides $p^{ns'} - 1$ with $\phi(m') \geq M$. As long as $s' \leq$polylog$(L \cdot \lambda)$ then we still have only a polylog "embedding overhead", and m' may be much smaller than $m = p^{ns} - 1$. Unfortunately we were not able to prove that such $s' \leq$polylog$(L \cdot \lambda)$ and $m' \leq \tilde{O}(L \cdot \lambda)$ always exist, we consider this an interesting open problem.

Approach 2: Using Prime Fields. An alternative, simpler, approach is to just pick $m = p' - 1$ for a prime number p' sufficiently larger than M, (so as to get $\phi(m) \geq M$), and set our "embedded plaintext space" to be $\mathbb{F}_{p'}$. This will give us the "simple case" that we discussed earlier in this section, where Φ_m factors into linear terms mod p'. Note that in this case we clearly have $m = \tilde{O}(M)$, so (a) the "embedding overhead" is at most $O(\log M) = \tilde{O}(\log(L\lambda))$, and (b) as long as we work with circuits of width $\tilde{\Omega}(\lambda)$ we can pack enough plaintext elements into each ciphertext to get low overhead.

This solutions has a few drawbacks, however. One relatively minor drawback is that the native operations of the scheme are now over a characteristic-p' field, and if $p' > p$ then the bound M on the dimension will be slightly larger than before (since the noise in fresh ciphertexts is now of the form $p' \cdot e$ rather that $p \cdot e$). A more serious problem is that each gate of the underlying circuit must now be emulated using a polynomial mod p'. We note, however, that this only results in a logarithmic slowdown: It is not hard to see that arithmetic over $\mathbb{F}_{p^n}$ can be emulated by mod-p' circuits of depth and size $O(n \cdot \log p)$ (e.g., express these operations as binary circuits and emulate that binary circuit mod-p').

Once we determined the parameter m and the "embedded plaintext space", all the other parameters of the scheme easily follow, and we obtain the following theorem:

Theorem 3. *For security parameter λ, any t-gate, depth-L arithmetic circuit of average width $\Omega(\lambda)$ over underlying plaintext space $\mathbb{F}_{p^n}$ (with $p^n \leq poly(\lambda)$) can be evaluated homomorphically in time $t \cdot \tilde{O}(L) \cdot polylog(\lambda)$.*

4.4 Achieving Depth-Independent Overhead

Theorem 3 implies that we can implement shallow arithmetic circuit with low overhead, but when the circuit gets deeper the dependence of the overhead on L causes the overhead to increase. Recall that the reason for this dependence on the depth is that in the BGV cryptosystem [3], the moduli get smaller as we go up the circuit, which means that for the first layers of the circuit we must choose moduli of bitsize $\Omega(L)$.

As explained in [3], the dependence on the depth can be circumvented by using bootstrapping. Namely, we can start with a modulus which is not too large, then reduce it as we go up the circuit, and once the modulus become too small to do further computation we can bootstrap back into the larger-modulus ciphertexts, then continue with the computation.

For our purposes, we need to ensure that we bootstrap often enough to keep the moduli small, and yet that the time we spend on bootstrapping does not significantly impact the overhead. Here we apply to the analysis from [3], that shows that a packed ciphertext with $\tilde{\Omega}(\lambda)$ slots can be decrypted using a circuit of size $\tilde{O}(\lambda)$ and depth polylog(λ). Hence we can even bootstrap after every layer of the circuit and still keep the overhead polylogarithmic, and the moduli never grow beyond polylogarithmic bitsize. We thus get:

Theorem 4. *For security parameter λ, any t-gate arithmetic circuit of average width $\Omega(\lambda)$ over underlying plaintext space $\mathbb{F}_{p^n}$ (with $p^n \leq poly(\lambda)$) can be evaluated homomorphically in time $t \cdot polylog(\lambda)$.*

References

1. Paul T. Bateman, Carl Pomerance, and Robert C. Vaughan. On the size of the coefficients of the cyclotomic polynomial. In *Topics in Classical Number Theory, Vol. I*, pages 171–202, 1984.
2. Václav E. Beneš. Optimal rearrangeable multistage connecting networks. *Bell System Technical Journal*, 43:1641–1656, 1964.
3. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Manuscript at http://eprint.iacr.org/2011/277, 2011.
4. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE, 2011.
5. Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.
6. I. Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarais. Multiparty computation from somewhat homomorphic encryption. Manuscript at http://eprint.iacr.org/2011/535, 2011.
7. Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 445–465. Springer, 2010.

8. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `http://crypto.stanford.edu/craig`.
9. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.
10. Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011.
11. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 2006.
12. Kristin Lauter, Michael Naehrig, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? Manuscript at http://www.codeproject.com/News/15443/Can-Homomorphic-Encryption-be-Practical.aspx, 2011.
13. Frank Thomson Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*. M. Kaufmann Publishers, 2 edition, 1992.
14. G. Lev, N. Pippenger, and L. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, C-30:93–100, 1981.
15. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, 2010.
16. Ron Rivest, Leonard Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–180, 1978.
17. Nigel P. Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography - PKC'10*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010.
18. Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. Manuscript at http://eprint.iacr.org/2011/133, 2011.
19. Damien Stehlé and Ron Steinfeld. Faster fully homomorphic encryption. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 377–394. Springer, 2010.
20. Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, 1968.
21. Lawrence C. Washington. *Introduction to Cyclotomic Fields*, volume 83 of *Graduate Texts in Mathematics*. Springer, 1996.

A Additional Optimizations

A.1 Faster Cloning

In Lemma 5 we establish that we can clone w' values using ℓ-fold operations in time $O((w' \log w')/\ell)$. Below we show how to remove the $\log w'$ term, which would allow us to clone values between levels in the circuit using asymptotically optimal $O(w'/\ell)$ time.

Recall that for the cloning procedure we are given a "multi-array" $\mathbf{A}'$ consisting of several ℓ-element arrays, and also the intended multiplicities of the values in these arrays $m_1, \ldots, m_w$. As before, denote the maximum intended multiplicity by $M = \max_i\{m_i\}$. The new procedure consists of two main parts:

Decomposition: For $i = 0, 1 \ldots, M$, construct a "multi-array" $\mathbf{A'}_i$ that contains the elements whose intended multiplicity is at least 2^i, as follows:

Set $\mathbf{A'}_0 = \mathbf{A}'$. Then for $i > 0$ we compute $\mathbf{A'}_i$ from $\mathbf{A'}_{i-1}$ by marking the slots of all the elements with intended multiplicity smaller than 2^i as empty, and then merging sparse arrays until the multi-array is at least half-full (or contains only one array). Note that when computing $\mathbf{A'}_i$ from $\mathbf{A'}_{i-1}$, we also keep a copy of $\mathbf{A'}_{i-1}$ for use in the aggregation part below.

Aggregation: For $i = M, \ldots, 1, 0$, construct a multi-array $\mathbf{A}_i$ as follows. Set $\mathbf{A}_M = \mathbf{A'}_M$, then for all $i < M$ concatenate two copies of $\mathbf{A}_{i+1}$ with one copy of $\mathbf{A'}_i$, and if the result is not half full them merge sparse arrays until it is half full again. The result is $\mathbf{A}_i$.

Note since each of $\mathbf{A}_{i+1}, \mathbf{A'}_i$ is either half full or contains a single array, then at most two merge operations are needed in each aggregation step. The output of the cloning procedure is $\mathbf{A}_0$.

Lemma 9. *The procedure above is correct, and it uses only $O(\frac{w'}{\ell} + \log w')$ copy and merge operations on ℓ-element arrays, where $w' = \sum_i m_i$*

Proof. Consider an arbitrary element of the input multi-array $\mathbf{A}'$, with intended multiplicity $m_i \in [2^j, 2^{j+1}-1]$ for some j. The decomposition part will output multi-arrays such that this element is in each of $\mathbf{A'}_0, \ldots, \mathbf{A'}_j$. Then, during the aggregation part, $\mathbf{A}_j$ will include one copy of this element, $\mathbf{A}_{j-1}$ three copies, $\mathbf{A}_{j-2}$ seven copies, and in general $\mathbf{A}_{j-k}$ contains $2^{k+1}-1$ copies. Hence at the end of the aggregation part, $\mathbf{A}_0$ includes $2^{j+1}-1$ occurrences of this element (which is at least as much as m_i but less than $2m_i$).

To analyze complexity, notice that the number of arrays in every multi-array $\mathbf{A'}_j$ equals the number of arrays in $\mathbf{A'}_{j-1}$ minus the number of merge operations that were used when computing $\mathbf{A'}_j$. Since $\mathbf{A'}_M$ cannot have less than zero arrays, it follows that the total number of merge operations throughout the decomposition part cannot be more than the initial number of arrays, namely $\lceil 2w/\ell \rceil \leq \lceil 2w'/\ell \rceil$. We observed above that the aggregation part does at most two merges for each $\mathbf{A}_j$, so the total number of merges during this part is at most $2\lceil \log M \rceil \leq 2\lceil \log w' \rceil$. Thus the total number of merge operations is bounded by $N = \lceil 2w'/\ell \rceil + 2\lceil \log M \rceil = O(\frac{w'}{\ell} + \log w')$.

Finally, the output multi-array $\mathbf{A'}$ contains at most twice as many occurrences of each element as needed, and it is at least half full. Hence it contains at most $\lceil \frac{4w'}{\ell} \rceil$ arrays, which means that the entire procedure duplicated arrays at most $\lceil \frac{4w'}{\ell} \rceil + N = O(\frac{w'}{\ell} + \log w')$ times. □

The procedure above can be made particularly efficient in our case, when used in conjunction with the following optimization: When considering a circuit, we sort the gates in each level according to their fan-out, thus making the input to the cloning procedure sorted by the intended multiplicity. Note that the decomposition part now becomes unnecessary, we just define $\mathbf{A'}_j$ to be the collection of the first few arrays, all the ones that contain elements of intended multiplicity at least 2^j.

Also important is that once the inputs are sorted, merging arrays do not need the full power of the Permute operation. As long as we keep the full slots in the arrays continuous, we can use the simple rotation operation to align the two arrays before we merge them. (The same can be done with the "higher-dimensional rotations" that we get in the general case in Section 4.) Hence the entire cloning network can be implemented using only $O(\frac{w'}{\ell} + \log w')$ basic operations of ℓ-Add, ℓ-Mult, and $\mathcal{H}$-actions.

A.2 Faster Routing

Tracing through the proofs in Section 2, in conjunction with the more efficient cloning technique from above, one can verify that the $\log W$ term in the statement of Theorem 1 can be made to multiply only the number of ℓ-Add and ℓ-Mult gates, not ℓ-Permute, which can make a big difference in practice. Roughly, the $\log W$ term arises from the fact that we seem to need $\Omega(W \cdot \log W)$ computation (in the worst-case) to route the inter-level wires. Note that such a $\log W$ term does not appear in the overhead of non-batched FHE schemes that operate on singletons rather than arrays. It seems plausible that this term could be eliminated somehow, and we consider this an interesting open problem.

A.3 Powering (Almost) for Free

In some applications, plaintext elements are not bits or integers, but rather elements in a finite extension field. For example, when implementing homomorphic AES, it may be convenient to use $\mathbb{F}_{2^8}$ as the underlying plaintext space [12, 18]. In these cases, the corresponding Galois group (whose automorphisms we use to permute the slots) includes also the Frobenius automorphism. (This is $x \to x^{2^j}$ in the AES example, and more generally $x \to x^{p^j}$

when using a characteristic-p field.) We show in Section 4 that applying the Galois group transformations to packed ciphertexts results in almost no additional noise. Thus we get a new function, ℓ-Frobenius, that raises the ℓ slots in parallel to a power of p, while adding almost no additional noise. This may not be surprising, since the Frobenius map is a linear operation on $\mathbb{F}_{p^n}$.

In practice this turns out to be a useful optimization for particular functions of interest: For the case of AES, the only non-linear part of AES is inversion in $\mathbb{F}_{2^8}$, which is equivalent to exponentiation to the 254-th power. While this may seem to be high-degree, the Frobenius automorphism allows us to evaluate this power relatively cheaply on ℓ elements in parallel. For an $a \in \mathbb{F}_{2^8}$ sitting in a plaintext slot, we use the Frobenius map to compute $a_j = a^{2^j}$ for $j = 1, 2, \ldots, 7$ (these are the '1's in the binary representation of 254), then multiply all the a_j to get $a^{254} = a^{-1}$. Thus, we can evaluate a^{254} at a price of only seven products (in terms of noise), and this 7-fold product can be computed by a depth-3 circuit. The binary affine transformation of the AES S-box is not linear over $\mathbb{F}_{2^8}$, but it *is* linear over the outputs of the Frobenius automorphisms, and so it *is* linear in terms of its effect on ciphertext noise (although to extract and pack the bits uses up two more levels in the circuit). The ShiftRows and MixColumns operation take four more levels using our permutation networks, and the matrix multiplication in the MixColumns uses another level. An AES round can therefore be accomplished using only a depth-10 circuit (in terms of noise), so homomorphic implementation of the full AES-128 will take a circuit of depth less than 100. It is therefore plausible that we could implement AES-128 homomorphically without resorting to bootstrapping at all!!! (We note, however, that many other optimizations are possible, and it is not clear if the approach sketched above is really the most efficient one for implementing AES-128.)

B Proofs

Lemma 1. *Let $S = \{0, \ldots, a-1\} \times \{0, \ldots, b-1\}$ be a set of ab positions, arranged as a matrix of a rows and b columns. For any permutation π over S, there are permutations π_1, π_2, π_3 such that $\pi = \pi_3 \circ \pi_2 \circ \pi_1$ (that is, π is the composition of the three permutations) and such that π_1 and π_3 only permute positions within each column (these permutations only change the row, not the column, of each element) and π_2 only permutes positions within each row. Moreover, there is a polynomial-time algorithm that given π outputs the decomposition permutations π_1, π_2, π_3.*

Proof. The basic strategy of the decomposition is that π_2 will send each element to some address with the same y-coordinate as its target destination, and similarly π_3 will correct all of the x-coordinates. The permutation π_1, on the other hand, serves as a strategic indirection. The reason this indirection is needed – i.e., the reason we cannot decompose π just as $\pi_3 \circ \pi_2$ with the properties above – is that several elements in the same row could have the same target y-coordinate (and thus π_2 cannot achieve its goal). Thus, π_1 is used to ensure that, when π_2 receives its input, no two elements in the same row have the same target column. The only nontrivial part of the proof is showing that a suitable π_1 always exists.

For $s \in S$, let s_x and s_y denote its x and y coordinates, namely $s = (s_x, s_y)$. Consider a bipartite graph $G = (V_1, V_2, E)$ where V_1 and V_2 each have b vertexes with labels $\{0, \ldots, b-1\}$. For every $s \in S$, we draw an edge from the V_1-vertex labeled s_y to the V_2-vertex labeled $\pi(s)_y$, and we label the edge 's'. (We may have more than one edge between the same pair of vertices's.) Clearly, this is a bipartite, a-regular graph. Therefore G's edges can be partitioned into a perfect matches, and this partition can be computed efficiently (e.g., using network-flow algorithms). In other words, one can compute in polynomial time a coloring of the edges of G using the colors $\{0, \ldots, a-1\}$, such that for all i the i-colored subgraph G_i of G is a perfect matching.

Let $\rho(s)$ denote the color of the edge labeled 's'. Now, define π_1, π_2, π_3 as follows: for all $s = (s_x, s_y) \in S$:

$$\pi_1(s) = (\rho(s), s_y), \quad \pi_2 \circ \pi_1(s) = (\rho(s), \pi(s)_y), \quad \pi_3 \circ \pi_2 \circ \pi_1(s) = (\pi(s)_x, \pi(s)_y)$$

Clearly, π_1, π_3 have the claimed property of only permuting within columns and π_2 only permutes within rows. All that remains is to establish that they are all well-defined permutations – i.e., that no "collisions" occur. π_1 is a permutation because no two edges emanating from the V_1-vertex labeled 's_y' have the same color. π_2 is a permutation, in particular it permutes elements in row i, because the subgraph G_i is a perfect matching. Finally, π_3 is a permutation since both $\pi_2 \circ \pi_1$ and π are permutations and since $\pi = \pi_3 \circ \pi_2 \circ \pi_1$. □

Lemma 4. *Evaluating ℓ permutation networks in parallel, each permuting k items, can be accomplished using $O(k \cdot \log k)$ gates of ℓ-Add and ℓ-Mult, and depth $O(\log k)$. Also, evaluating a permutation π over $k \cdot \ell$ elements that are packed into k ℓ-element arrays, can be accomplished using k ℓ-Permute gates and $O(k \log k)$ gates of ℓ-Add and ℓ-Mult, in depth $O(\log k)$. Moreover, there is an efficient algorithm that given π computes the circuit of ℓ-Permute, ℓ-Add, and ℓ-Mult gates that evaluates it, specifically we can do it in time $O(k \cdot \ell \cdot \log(k \cdot \ell))$.*

Proof. The first statement follows directly from Lemma 3 and the discussion above. The second statement follows from Lemma 1, which says that the permutation π can be decomposed as $\pi = \pi_3 \circ \pi_2 \circ \pi_1$ where π_1 and π_3 each involve evaluating n permutation networks in parallel across the ℓ indexes, and π_2 only permutes elements within each ℓ-element array, and therefore can be done using k gates of ℓ-Permute and just one level.

The efficiency of computing the circuit that realizes π follows from the fact that the decomposition π_1, π_2, π_3 can be computed efficiently, as per Lemma 1. In fact, it was shown by Lev et al. [14] that this decomposition can be computed in time $O(k \cdot \ell \cdot \log(k \cdot \ell))$. □

Lemma 5. ***(i)*** *The cloning procedure from Figure 1 is correct.*

(ii) *Assuming that at least half the slots in the input arrays are full, this procedure can be implemented by a network of $O(w'/\ell \cdot \log(w'))$ ℓ-fold gates of type ℓ-Add, ℓ-Mult and ℓ-Permute, where w' is the total number of full slots in the output, $w' = \sum m_i$. The depth of the network is bounded by $O(\log w')$.*

(iii) *This network can be constructed in time $\tilde{O}(w')$, given the input arrays and the m_i's.*

Proof. In each phase j, first the number of occurrences of every value is doubled, and next if a value v_i occurs more than m_i times then the excess occurrences are removed. Therefore after the j'th phase each value v_i is duplicated $\min(m_i, 2^j)$ times. Denoting the number of full slots after the j'th phase by $w_j \stackrel{\text{def}}{=} \sum_i \min(m_i, 2^j)$, we have at the end of phase j some number k_j of ℓ-slot arrays, where $(k_j - 1)\ell/2 < w_j \leq k_j \cdot \ell$, since once the merging part is over we must have at least half the slots full. Correctness now follows easily just by looking at $j = \lceil \log M \rceil$.

Regarding complexity (part (ii)), we note that if the input arrays are at least half full then at the beginning of every iteration we have $k_{j-1} \leq 2w_{j-1}/\ell =< 2w'/\ell = O(w'/\ell)$ arrays (clearly $w_j < w'$ for all j by definition.) After the duplication step (Line 2) we have $2k_{j-1}$ arrays, and then each merging step (Line 6) removes one array, so we can have at most $2k_{j-1} = O(w'/\ell)$ such steps. Observing that every merge takes a constant number of gates (two ℓ-Permute gates and one Select operation), we conclude that each phase takes at most $O(w'/\ell)$ ℓ-fold gates.[7] The number of phases is $\lceil \log M \rceil \leq \lceil \log w' \rceil$, and the claimed complexity follows.

Part (iii) follows easily by noting that the network implementing each phase can be constructed in time quasi-linear in the number of slots that are available at the beginning of that phase, just by using greedy algorithms to make all the decisions. (The most time-consuming operation is marking entries as "don't-care"s in Line 4, everything else can be done in time $\tilde{O}(w'/\ell)$.) □

Theorem 1. *Let ℓ, t, w and W be parameters. Then any t-gate fan-in-2 arithmetic circuit C with average width w and maximum width W, can be evaluated using a network of $O\big(\lceil t/\ell \rceil \cdot \lceil \ell/w \rceil \cdot \log W \cdot \text{polylog}(\ell)\big)$ ℓ-fold gates*

[7] Note that removing redundant values (Line 4) does not take any gates, we leave the arrays unchanged and just mark the redundant values as "don't-care"s.

*of types ℓ-*Add*, ℓ-*Mult*, and ℓ-*Permute*. The depth of this network of ℓ-fold gates is at most $O(\log W)$ times that of the original circuit C, and the description of the network can be computed in time $\tilde{O}(t)$ given the description of C.*

Proof. Consider one level of the circuit with w' gates, where in the previous level we computed $w \leq 2w'$ input values, packed into $O(\lceil w/\ell \rceil)$ ℓ-element arrays. Our approach is to first clone and then permute these values so that the $2w'$ input slots of the w' gates are filled correctly. More precisely, these $2w'$ input slots will be arranged in two sets of ℓ-slot array, one set for the left inputs and the other for the right inputs to all the gates. Concatenating these two sets of arrays into two multi-arrays, we arrange the slots such that the left and right inputs to each gate are aligned in the same index in the two multi-arrays. Once all the values are routed to their correct locations in the multi-arrays, the actual computation of the gates in this layer can obviously be evaluated only $O(\lceil w'/\ell \rceil)$ ℓ-fold gates of ℓ-Adds or ℓ-Mults.

By Lemma 5, we can compute the multi-arrays of $O(w'/\ell)$ ℓ-element arrays that contains the inputs with sufficient multiplicity using $O(\lceil w'/\ell \rceil \cdot \log(w'))$ ℓ-fold gates. The resulting multi-arrays have $O(w)$ slots (more than either the source or target multi-arrays), at least half of which contain "real values" while the other slots contain "don't-care"s. Let π be a permutation over these $O(w)$ slots that maps the slots that contain the real values to the appropriate positions in the target multi-arrays. By Lemma 4 we can evaluate π with a network of $O(w'/\ell \text{polylog} \lceil w'/\ell \rceil)$ n-fold gates, and can compute the structure of that network in time $\tilde{O}(w')$.

The result for the whole circuit follows easily, using as our inductive hypothesis that the w' outputs are indeed packed into $O(\lceil w'/\ell \rceil)$ ℓ-element arrays for input to the next level. □

Lemma 6. *Fix an integer ℓ and let $k = \lceil \log \ell \rceil$. Any permutation π over $I_\ell = \{0, \ldots, \ell - 1\}$ can be implemented by a $(2k-1)$-level network, with each level consisting of a constant number of rotations and* Select *operations on ℓ-arrays.*

Moreover, regardless of the permutation π, the rotations that are used in level i $(i = 1, \ldots, 2k-1)$ are always exactly $2^{|k-i|}$ and $\ell - 2^{|k-i|}$ positions, and the network depends on π only via the bits that control the Select *operations. Finally, this network can be constructed in time $\tilde{O}(\ell)$ given the description of π.*

Proof. If ℓ is a power of two then the network is just a Beneš network. Otherwise (i.e., $2^{k-1} < \ell < 2^k$ for some k) the basic strategy is to realize a permutation over I_ℓ by using two k-element arrays to realize a Beneš permutation network over the first 2^k of the 2ℓ positions. We realize each level of the Beneš network using a constant number of rotations and Select operations. Since $2^k > \ell$ then clearly any permutation on I_ℓ can be expressed as a permutation over the first 2^k positions (e.g., where the last $2^k - \ell$ elements remain fixed).

It remains only to show how to realize an i-offset-swap over the first 2^k elements using just a constant number of operations on the two ℓ-slot arrays. Clearly, we can handle all the pairs $(v, v+j)$ where both indexes are in the same array using the rotations j and $\ell - j$ and two Select operations, applied to the each of the arrays. To handle the pairs where v is in the first array and $v+j$ is in the second (at index $v+j-\ell$), we shift the first array by $\ell - j$ and the second array by j, then again use two Select operations (one Select on the first array and the shifted version of the second, the other Select on the second array and the shifted version of the first). All in all we have four rotation operations (two for each array) and six Select's. The "Finally" part follows directly from Lemma 3. □

Lemma 8. *For all positive integers m we have $m/\phi(m) = O(\log \log m)$.*

Proof. The "worst-case" that maximizes $m/\phi(m)$ is when m is a product of distinct primes $m = p_1 \cdots p_t$, in which case we have $m/\phi(m) = p_1/(p_1 - 1) \cdots p_t/(p_t - 1)$. Clearly, the worst-case is when the p_i's are the *first* t primes. In this case, we can use the prime number theorem to argue that $p_t = \text{polylog}(m)$ (actually, something like $\log m$). By Merten's theorem the product over primes $\prod_{p < \text{polylog} m} p/(p-1)$ is $\theta(\log \log m)$.

C Basic Algebra

To understand our techniques it is first necessary to recap on the underlying algebra of cyclotomic fields. We have tried to cover as much detail as needed, but the reader should be aware a self contained treatment will be hard to come by in such a short space. We therefore refer the interested reader to [21] for details on cyclotomic fields.

C.1 Reductions of Cyclotomic Fields

We let $\Phi_m(X)$ be the m-th cyclotomic polynomial, and let $K = \mathbb{Q}(\zeta_m)$ denote the associated number field. The degree of Φ_m is $\phi(m)$, where $\phi(\cdot)$ is Euler's phi-function. Note that asymptotically m is of the same size as $\phi(m)$, but for the small values of m that we will use in practice, $\phi(m)$ is roughly 10%-50% smaller than m. We associate K with the set of rational polynomials in X of degree less than N, with multiplication and addition defined modulo Φ_m. We let the ring of integers of K be denoted by $\mathcal{O}_K = \mathbb{Z}[\zeta_m]$.

We now fix a prime p, which is neither ramified in K, nor an index divisor (i.e. p does not divide m). Consider the reduction of K at p; we define

$$\mathbb{A}_p := \mathbb{Z}_p[X]/\Phi_m(X)$$

to be the ring of polynomials over $\mathbb{Z}_p$ where multiplication and addition are defined modulo Φ_m and p. Note, we assume that the representation of $\mathbb{A}_p$ is such that the coefficients are given in the range $(-p/2, p/2]$. In general $\mathbb{A}_p$ is not a field but is an *algebra*, since Φ_m is generally not irreducible mod p.

Since p is neither an index divisor nor ramified, and because $K/\mathbb{Q}$ is Galois, we have that the polynomial Φ_m splits mod p into ℓ distinct factors $F_i(X)$, each of degree d, where $\ell \cdot d = \phi(m)$. We then have that

$$\begin{aligned}\mathbb{A}_p &\cong \mathbb{Z}_p[X]/F_0(X) \times \ldots \times \mathbb{Z}_p[X]/F_{\ell-1}(X)\\ &= \mathbb{L}_0 \times \ldots \times \mathbb{L}_{\ell-1} =: A_p.\end{aligned}$$

i.e. the reduction of K modulo p is isomorphic to ℓ copies $\mathbb{L}_i = \mathbb{Z}_p[X]/F_i(X)$ of $\mathbb{F}_{p^d}$. Since all finite fields of a given degree are isomorphic, each of these copies of $\mathbb{F}_{p^d}$ is isomorphic to each other. Note we let $\mathbb{A}_p$ denote the representation of the algebra by polynomials modulo Φ_m and A_p denote the algebra by a set of l copies of the fields defined by the polynomials $F_i(X)$.

We note there is a natural homomorphic inclusion maps $\mathbb{A}_p \longrightarrow \mathcal{O}_K$ defined by mapping $\mathbb{A}_p$ to the coset representative with coefficients in $(-p/2, p/2]$. If $\alpha \in \mathcal{O}_K$ then we let $\alpha \mod p$ denote the inverse in $\mathbb{A}_p$ under this inclusion. If q is a prime greater than p then we can also consider elements of $\mathbb{A}_p$ as elements in $\mathbb{A}_q$ but this inclusion is not a homomorphism (since it only preserve the arithmetic operations "as long as there is no wraparound").

We will use $\mathbb{A}_p$ (resp. A_p) in two distinct ways. In the first way we use $\mathbb{A}_p$ and A_p to describe the message space of our scheme; in this case we take p to be small (think $p = 2$, or a 32-bit prime). In the second way, we use $\mathbb{A}_q$ (for a large prime q) as an approximation of the global object $\mathbb{A}$. Looking ahead the basic construction is that we take an element $\alpha \in \mathbb{A}_p$, then form the element in $\mathbb{A}_q$ given by $\alpha + p^t \cdot \tau$, where τ is referred to as the *noise*. Public operations are then performed, and these will correspond to valid operations in $\mathbb{A}_p$ only if the noise term does not become too large (in the sense of the ∞-norm of the noise becoming bigger than $q/2$). If the operation is does not result in wrap-around then we can (upon decrypting) obtain the plaintext in $\mathbb{A}_p$.

C.2 Underlying Plaintext Algebra

Each message in $\mathbb{A}_p$ actually corresponds to ℓ messages in $\mathbb{F}_{p^d} \cong \mathbb{Z}_p[X]/F_i(X)$. We call each of these components a "slot". By the Chinese Remainder Theorem, additive and multiplicative operations in $\mathbb{A}_p$ correspond to SIMD

operations on the slots. However, in many applications we will be interested in plaintexts where each slot lies in $\mathbb{F}_{p^n}$, for some n dividing d. (In particular this includes the important case of $n = 1$.) In addition an application may have a preferred representation (i.e. preferred polynomial basis) for the underlying field, $\mathbb{F}_{p^n}$.

We therefore fix (or are given) an irreducible polynomial $G(X) \in \mathbb{Z}_p[X]$, of degree n, which defines the specific polynomial basis we are interested in; we take $G(X) = X - 1$ when $n = 1$. To fix notation we define $\mathbb{K}_n = \mathbb{Z}_p[X]/G(X)$ to denote one copy of this degree n field, with the given polynomial representation.

Note, in applications one is given p and n, and then one needs to find values of m which enable the above representation. Basic algebra shows us that $\Phi_m(X)$ will have a degree d factor if and only if m divides $p^d - 1$. Thus, given p and n, we need to select m such that for some value $d = s \cdot n$, we have m divides $p^d - 1$. The value ℓ is given by $\phi(m)/d$.

For each of our fields $\mathbb{L}_i = \mathbb{Z}_p[X]/F_i(X)$ there will be a distinct homomorphic embedding of $\mathbb{K}_n$ into $\mathbb{L}_i$ which we will denote by $\Psi_{n,i}$, which will be an isomorphism in the case when $n = d$. Our basic plaintext space will now be defined as ℓ copies of $\mathbb{K}_n$, i.e. $\mathcal{M} = (\mathbb{K}_n)^\ell$, where addition and multiplication will be defined component-wise. We therefore can define a map

$$\Psi_n : \begin{cases} \mathcal{M} & \longrightarrow & A_p \\ (m_0, \ldots, m_{\ell-1}) & \longmapsto & (\Psi_{n,0}(m_0), \ldots, \Psi_{n,\ell-1}(m_{\ell-1})). \end{cases}$$

By applying the Chinese Remainder Theorem given an element $\mathbf{a} \in A_p$ we can obtain a value $\alpha \in \mathbb{A}_p$; we write $\alpha = \mathsf{CRT}_p(\mathbf{a})$. Note, our use of notations: Elements in $\mathbb{A}_p$ and $\mathbb{B}_p$ will be represented by lower case Greek letters; elements in A_p and $\mathcal{M}$ will be represented by bold face roman letters (since they are vectors); and elements in $\mathbb{K}_n$ and $\mathbb{L}_i$ will be represented by standard lower case roman letters.

We end this discussion of the plaintext space by noting that there is a simple operation that produces the projection map. If we consider the element $\pi_i \in \mathbb{A}_p$ which is defined by the element in A_p given by the i unit vector $\mathbf{e}_i$. Then if $\mathbf{m} = (m_0, \ldots, m_{\ell-1}) \in A_p$ that $\pi_i \cdot \mathsf{CRT}_p(\mathbf{m}) = \mathsf{CRT}_p(0, \ldots, 0, m_i, 0, \ldots, 0)$. From π_i we can also define a projection on an arbitrary subset $I \subset \{0, \ldots, \ell - 1\}$ in the obvious way; by defining π_I to be the element $\sum_{i \in I} \mathsf{CRT}_p(\mathbf{e}_i)$.

C.3 Galois Theory of Cyclotomic Fields

The field $K = \mathbb{Q}(\zeta_m)$ is abelian (i.e. has abelian Galois group) and has Galois group given by $\mathcal{G}\mathsf{al}(K/\mathbb{Q}) \cong (\mathbb{Z}/m\mathbb{Z})^*$. If we think of X in the representation of K as denoting a generic mth root of unity ζ_m, then given an element $i \in (\mathbb{Z}/m\mathbb{Z})^*$ the associated element of the Galois group is given by the mapping $\kappa_i : X \mapsto X^i$.

We now need to consider how the Galois group $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ works when we consider K modulo p, to A_p and $\mathbb{A}_p$. Notice, that since $\mathbb{A}_p$ is not a field the usual theorems of Galois Theory do not apply (an obvious fact but worth stating). The maps defined by the Galois group commute with our functions Ψ_n, and CRT_p etc. Thus, to fix ideas, consider an element $\mathbf{m} = (m_0, \ldots, m_{\ell-1}) \in \mathcal{M} = \mathbb{K}^\ell$. We obtain the corresponding element in $\mathbb{A}_p$ by applying $\alpha = \mathsf{CRT}_p(\Psi_n(\mathbf{m})) \in \mathbb{A}_p$. Now if we apply the element κ_i from $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ to the element α we obtain an element β such that $\beta = \mathsf{CRT}_p(\Psi_n(\kappa_i(m_1), \ldots, \kappa_i(m_\ell)))$, where $\kappa_i(m_j(X)) = m_j(X^i) \pmod{G(X)}$.

Considering how automorphisms work on $\mathbb{A}_p$, it is well known that any field $\mathbb{F}_{p^k}$ has Galois group over $\mathbb{Z}_p$ given by the cyclic group C_k of order k. Now since $\mathbb{A}_p$ contains the subfield $\mathbb{F}_{p^d}$ we have that $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ contains the cyclic subgroup $C_d \lhd (\mathbb{Z}/m\mathbb{Z})^*$. The group C_d is called the *decomposition group* of a prime ideal lying above p in K. The group C_d is generated by the element $p \in (\mathbb{Z}/m\mathbb{Z})^*$, which corresponds to the Frobenius map $\kappa_p : X \mapsto X^p$. In what follows we let $\mathcal{G}$ denote this subgroup C_d of $(\mathbb{Z}/m\mathbb{Z})^*$

Considering how $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ acts on $\mathbb{K}_n$, we notice that the Galois group of $\mathbb{K}_n$ over $\mathbb{Z}_p$ is given by $C_n \cong C_d/C_{d/n}$ and generated by the Frobenius map. The key difference, between $\mathbb{K}_n$ and $\mathbb{K}_d$, being that the map κ_{p^n}

is the identity on the subfields $\mathbb{K}_n$. If we want to restrict to the Galois group of $\mathbb{K}_n$ we let $\hat{\mathcal{G}}$ denote the subset $\{1, p, p^2, \ldots, p^{n-1}\}$ consisting of a set of representatives for the Galois group of $\mathbb{K}_n$.

Since $(\mathbb{Z}/m\mathbb{Z})^*$ is abelian all subgroups are normal, and hence we can define quotient groups, and so we define $\mathcal{H}$ to be the quotient group $(\mathbb{Z}/m\mathbb{Z})^*/\mathcal{G}$, note $\mathcal{H}$ has order ℓ. We write $\mathcal{H}$ as a product of cyclic groups $C_{n_1} \times C_{n_t}$ with n_i dividing n_{i+1}. As a set of coset representatives for $\mathcal{H}$ we first pick a coset representative h_i for C_{n_i}, and then as the coset representatives of all other elements we take those elements in $(\mathbb{Z}/m\mathbb{Z})^*$ given by

$$\prod_{i=1}^{t} h_i^{e_i} \text{ for } 0 \leq e_i < n_i.$$

Thus we can identify $\mathcal{H}$ with a *subset* of $(\mathbb{Z}/m\mathbb{Z})^*$.

If we label the roots of Φ_m in K by $\zeta_m^{(0)}$ to $\zeta_m^{(\phi(m)-1)}$ then it is a standard fact that the Galois group acts transitively on these roots. The subgroup $\mathcal{G}$ acts on these roots, and we can partition the set of roots into disjoint sets with respect to the group action of $\mathcal{G}$. That is we create $\ell = \phi(m)/d$ subsets each of d elements, we label these subsets $X_0, \ldots, X_{\ell-1}$. Since $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ acts transitively on the set $\{\zeta_m^{(0)}, \ldots, \zeta_m^{(\phi(m)-1)}\}$, the quotient group $\mathcal{H} = \mathcal{G}\mathsf{al}(K/\mathbb{Q})/\mathcal{G}$ acts transitively on the set $X_0, \ldots, X_{\ell-1}$.

Since $\mathcal{G}$ was the decomposition group of p the sets X_i, each containing d complex roots, when reduced modulo p can be placed in correspondence with the roots of $F_i(X)$, i.e. one of the factors of Φ_m modulo p. We need to fix a representative for for each set $X_i \mod p$. Fixing a representative for $X_i \mod p$ means essentially fixing a root of $F_i(X)$ modulo p; and one can think of the symbolic root X being such a root with all other roots being given by a polynomial in X modulo p of degree less than $d-1$ (when reduced arithmetic is considered modulo $F_i(X)$). Since $\mathcal{H}$ has order ℓ and acts transitively on $\{X_0, \ldots, X_{\ell-1}\}$, for each $i \in \{0, \ldots, \ell-1\}$ there is exactly one element σ_i in $\mathcal{H}$ which sends 0 to i. If we fix the representative of the set X_0 to be $\zeta_m^{(0)}$ then to define the representative of the set X_i we take $\sigma_i \in \mathcal{H}$ and set the representative of X_i to be $\sigma_i(\zeta_m^{(0)})$. Since, defining a representative of X_i essentially means fixing a representation of the field $\mathbb{Z}_p[X]/F_i(X)$ this then means that our set of representatives for $\mathcal{H}$ act "transitively on the plaintext slots" in the following sense: For each pair $i, j \in \{0, \ldots, \ell-1\}$ we have that

$$\sigma_j(\sigma_i^{-1}(\mathsf{CRT}_p(\Psi_n(0, \ldots, 0, m_i, 0, \ldots, 0)))) = \mathsf{CRT}_p(\Psi_n(0, \ldots, 0, m_j^{p^t}, 0, \ldots, 0)).$$

for some integer t. In the case $n = 1$ we have $m_j^{p^t} = m_j$ and so our set of representatives for $\mathcal{H}$ act directly as permutations on the slots.

Our main technical contribution in both practical and theoretical terms to FHE is based on the properties of the group $\mathcal{H}$ and how it acts on the plain text slots. It is clear, since $\mathcal{H}$ acts transitively as above and we have projection maps, that we can, given a vector of slots $(m_0, \ldots, m_{\ell-1}) \in \mathbb{K}_n^\ell$ map it to an arbitrary permutation of the slots. The naive algorithm for this, consisting of projecting each element, mapping via $\mathcal{H}$ as above, making sure we cope with the possibility of powering by Frobenius, and then recombining via addition, has complexity $O(\ell)$. In Section 3 we showed that an arbitrary permutation on the slots can be realized in $O(t \cdot \log \ell)$ operations, where t is the number of cyclic components of the group $\mathcal{H}$, note $t = O(\log \ell)$. That this algorithm can be applied in our case should be immediate, but to fix ideas, we examine how $\mathcal{H}$ acts on the slots when $\mathcal{H}$ is cyclic; and how to construct our offset swaps in this case.

When $\mathcal{H}$ is cyclic If $\mathcal{H} = \langle h \rangle$ is cyclic we can, by fixing on a given value of $F_0(X)$, reorder the factors $F_i(X)$ so that the factors are precisely those factors corresponding to $\sigma_h^i(1)$. Thus we can consider $\mathcal{H}$ as defining permutations on the factors of Φ_m modulo p. Although $\mathcal{H}$ is rarely cyclic this case is illustrative of what is occurring,

and in practice we can often restrict the number of slots to correspond to the largest cyclic subgroup of $\mathcal{H}$. [8] We consider three examples of increasing complexity:

Example 1: The simplest case to understand is when the decomposition group is trivial, i.e. $d = 1$. Consider the case of $m = 11$ and $p = 23$, we have that the polynomial $\Phi_m(X)$ factors into ten linear factors modulo 23, and the Galois group $(\mathbb{Z}/m\mathbb{Z})^*$ is cyclic of order 10 and generated by the element 2. Since $\mathcal{G} = \langle 1 \rangle$ we take using the procedure above $\mathcal{H} \cong (\mathbb{Z}/m\mathbb{Z})^* = \langle 2 \rangle$. Thus we have ten slots and we order them such that we have

$$\kappa_2(\mathsf{CRT}_p(\Psi_n(m_0, m_2, \ldots, m_9))) = \mathsf{CRT}_p(\Psi_n(m_9, m_0, m_2, \ldots, m_8)).$$

Hence κ_2 produces a cyclic shift of the slots. If we wish to switch elements in positions i and j, for $i < j$, the we only need to apply the following operation

$$\mathsf{swap}_{i,j}(\alpha) = \kappa_{2^{j-i}}(\pi_i \cdot \alpha) + \kappa_{2^{i-j}}(\pi_j \cdot \alpha) + \pi_{\{0,\ldots,9\}\setminus\{i,j\}} \cdot \alpha.$$

Example 2: To see what happens for non-trivial decomposition groups we consider the case of $m = 31$ and $p = 2$. We have since $2^5 \equiv 1 \pmod{31}$ that the decomposition group at p is cyclic of order 5, i.e. $d = 5$. In this example we find that by $\mathcal{G}\mathsf{al}$ factors directly into the product of $\mathcal{G} = \langle 2 \rangle$ and the cyclic subgroup $\langle 6 \rangle$. The set of coset representatives for $\mathcal{H}$ we can take to be this subgroup $\langle 6 \rangle$, thus we can identify $\mathcal{H}$ with a subgroup of $\mathcal{G}\mathsf{al}$. This implies that the elements in $\mathcal{H}$ act as direct permutations on the slots, and we do not need to worry about the action of Frobenius. In particular we can define the six slots so that we have, for a specific representation of $\mathbb{K}_n = \mathbb{F}_{2^5}$,

$$\kappa_6(\mathsf{CRT}_2(\Psi_n(m_0, m_1, m_2, m_3, m_4, m_5))) = \mathsf{CRT}_2(\Psi_n(m_5, m_0, m_1, m_2, m_3, m_4)).$$

If we wish to shift to the left we take the elements in $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ given by $1/6^i \pmod{m}$, so for example since $1/6 = 26 \pmod{31}$ we have

$$\kappa_{26}(\mathsf{CRT}_2(\Psi_n(m_0, m_1, m_2, m_3, m_4, m_5))) = \mathsf{CRT}_2(\Psi_n(m_1, m_2, m_3, m_4, m_5, m_0)).$$

If we wish to switch elements, for an element $\alpha \in \mathbb{A}_p$, in positions i and j, with $i < j$, then we apply the following operation

$$\mathsf{swap}_{i,j}(\alpha) = \kappa_{6^{j-i}}(\pi_i \cdot \alpha) + \kappa_{6^{i-j}}(\pi_j \cdot \alpha) + \pi_{\{0,\ldots,5\}\setminus\{i,j\}} \cdot \alpha.$$

Example 3: The above example, in which $\mathcal{H}$ could be identified with a subgroup of $\mathcal{G}\mathsf{al}$ is not typical. In the general case we have the added complication of dealing with actions of Frobenius on applying automorphism corresponding to elements in $\mathcal{H}$. We examine this more general situation via means of an example. We make $m = 257$ and $p = 2$. In this case we find that 2 has order 16 modulo m, and that the quotient group $\mathcal{H} = \mathcal{G}\mathsf{al}/\langle 2 \rangle$ is cyclic of order 16. We also find that there is no cyclic subgroup of order 16 of $\mathcal{G}\mathsf{al}$ which is not equal to $\langle 2 \rangle$. Thus $\mathcal{H}$ cannot be represented as a subgroup of $\mathcal{G}\mathsf{al}$.

We instead represent $\mathcal{H}$ by the set of coset representatives given by $3^i \bmod m$, for $i = 0, \ldots, 15$. Since $3^8 \bmod m = 136 \notin \langle 2 \rangle$, whilst $3^{16} \bmod m = 249 = 2^{11} \bmod m$. We therefore have 16 slots, each consisting of an element in $\mathbb{K}_n = \mathbb{F}_{2^{16}}$. We fix a specific representation of each slot so that

$$\kappa_3(\mathsf{CRT}_2(\Psi_n(m_0, m_1, \ldots, m_{14}, m_{15}))) = \mathsf{CRT}_2(\Psi_n(m_{15}^{2^{11}}, m_0, m_1, \ldots, m_{13}, m_{14})).$$

[8] For implementation purposes restricting the slots in this way is simpler, although for our asymptotic result on FHE with polylog overhead we will require to consider the whole of $\mathcal{H}$.

However, we also have

$$\kappa_8 6(\mathsf{CRT}_2(\Psi_n(m_0, m_1, \ldots, m_{14}, m_{15}))) = \mathsf{CRT}_2(\Psi_n(m_1, \ldots, m_{13}, m_{14}, m_{15}, m_0^{2^5})).$$

Note that $(1/3) \bmod m = 86$, but that 86 is not one of our coset representatives for $\mathcal{H}$.

In other words to move elements to the right (without wrap around) by i places we apply the map $\kappa_{3^i \bmod m}$, but to move elements to the left (without wrap around) by i places we need to apply the map $\kappa_{3^{-i} \bmod m}$. Hence if we wish to switch elements, for an element $\alpha \in \mathbb{A}_p$, in positions i and j, with $i < j$, then we apply the following operation

$$\mathsf{swap}_{i,j}(\alpha) = \kappa_{3^{j-i}}(\pi_i \cdot \alpha) + \kappa_{(1/3)^{j-i} \bmod m}(\pi_j \cdot \alpha) + \pi_{\{0,\ldots,5\} \setminus \{i,j\}} \cdot \alpha.$$

Hence, although the underlying algebra is different when $\mathcal{H}$ cannot be identified with a subgroup of $\mathcal{G}\mathsf{al}$, the method to obtain a swap is exactly the same.

These examples show that for cyclic groups we can realize any transposition via the use of scalar multiplication by the π_I and application of maps κ_i. The above technique also allows us to realize the offset swaps from Definition 1 for any subset $T \subset S = \{0, \ldots, \ell - 1\}$ and any i. The following technique works for when $\mathcal{H} = \langle h \rangle$ is a cyclic group generated by h, generalizing to other groups follows from our methods but leads to more complex formulas. Recall that a permutation π over S is an i-offset swap over S if there exists a subset $T \subset S$ such that the pairs $\{(t, t + i \bmod \ell) : t \in T\}$ are disjoint and π simply swaps each pair (leaving the other elements fixed).

For a set A we let $A + i = \{j + i \bmod \ell : j \in A\}$ and $\overline{A} = S \setminus A$. We also split T into two sets T_L and T_R such that $t \in T_L$ if and only if $t \in T$ and $t + i < \ell$, i.e. T_L is the set of elements in T which can be shifted to the left by i, without wrap around. Algebraically an offset swap on an element α is then defined in terms of our isomorphisms κ_i etc as

$$\pi_{\overline{T \cup (T+i)}} \cdot \alpha + \kappa_{h^i}(\pi_{T_L} \cdot \alpha) + \kappa_{(1/h)^i}(\pi_{T_L + i} \cdot \alpha) + \kappa_{(1/h)^{\ell - i}}(\pi_{T_R} \cdot \alpha) + \kappa_{h^{\ell - i}}(\pi_{T_R + i} \cdot \alpha)$$

The first term corresponds to those elements which are kept fixed by the offset swap, i.e. those elements neither in T nor $T + i$. The second term corresponds to those elements shifted to the left by i without wrap around, the third corresponds to elements shifted to the right by i without wrap around by i without wraparound, the final two terms deal with the case of wraparound.

D Using mod-Φ_m Polynomial Arithmetic

Part of our goal in this paper is to allow implementations of BGV-type cryptosystems over rings of the form $\mathbb{Z}[X]/\Phi_m(X)$ for arbitrary integers m, not only when m is a prime. Although most of the underlying algebra works the same way regardless of what m is, we do not have a good bound on the increase in the size of coefficient vectors when using mod-Φ_m arithmetic.

Recall that for every ring $\mathcal{R} = \mathbb{Z}[X]/F(X)$ there is a "ring-constant" $\gamma_{\mathcal{R}}$, such that for all $a, b \in \mathcal{R}$ it holds that $\|ab\| \leq \gamma_{\mathcal{R}} \cdot \|a\| \cdot \|b\|$, where $\|x\|$ is the norm of the coefficient-vector of x (say, the l_∞ norm). However, we do not have a good bound on the "ring-constant" for rings of the for $\mathcal{R}_m = \mathbb{Z}[X]/\Phi_m(X)$, and in particular $\gamma_{\mathcal{R}_m}$ can be super-polynomial in m. In particular $\gamma_{\mathcal{R}_m}$ is related to the sizes of the coefficients of $\Phi_m(X)$ which are known to get rather large [1]. In our context, this means that when multiplying two "short" ciphertexts, the result can be "longer" than the product of the two by this factor $\gamma_{\mathcal{R}_m}$ for which we do not have a good bound.

D.1 Canonical Embeddings and Norms

To analyze a cryptosystem that works mod-Φ_m, we therefore use a different measure of "size" of polynomials: Rather than considering the norm of the coefficient vector of a polynomial, we consider the norm of the "canonical embedding" of that polynomial: For an integer m, let $\mathcal{P}_m$ be the set of complex primitive m-th roots of unity. Then for a polynomial $a \in \mathbb{Q}[X]/\Phi_m(X)$, the "canonical embedding" of a is the vector of values that a assumes in all the roots in $\mathcal{P}_m$,

$$\mathcal{E}(a) \stackrel{\text{def}}{=} \left\langle a(\rho^k) \ : \ k \in \mathbb{Z}_m^* \right\rangle, \text{ where } \rho \text{ is a fixed complex primitive } m\text{-th root of unity (e.g., } \rho = e^{-2\pi i/m}).$$

More generally, the canonical embedding of an element $a \in \mathbb{Q}[X]/F(X)$ consists of the evaluations of a in all the complex roots of F. Below we only use the canonical embeddings for the cases $F(X) = \Phi_m(X)$ and $F(X) = X^m - 1$. Note that $\mathcal{E}(a)$ is in general a vector of complex numbers, and the size of each entry in that vector is the norm (absolute value) of that complex number.

Below we refer to the norm of $\mathcal{E}(a)$ as the "canonical embedding norm" of a, and denote it by $\|a\|^{can}$. Although it is possible to define the "canonical embedding l_p norm" for any l_p, below we always refer to the canonical embedding l_∞ norm. Namely,

$$\|a\|^{can} \stackrel{\text{def}}{=} \|\mathcal{E}(a)\| = \max_{k \in \mathbb{Z}_m^*} |a(\rho^k)|.$$

(Note again that in this section we consistently use $\|\cdot\|$ to refer to the l_∞ norm of a vector and not the l_2 norm.) We extend the canonical embedding norm to vectors over $\mathbb{Q}[X]/\Phi_m(X)$ in the natural way, namely if $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$ is an n-vector over $\mathbb{Q}[X]/\Phi_m(X)$, then $\|\mathbf{a}\|^{can} = \max_{i<n} \|a_i\|^{can}$.

It is easy to see that for any element $a \in \mathbb{Q}[X]/\Phi_m(X)$, the canonical embedding norm is not much more than the coefficient norm, namely $\|a\|^{can} < \phi(m) \cdot \|a\|$ (where $\|a\|$ is the norm of a's coefficient vector). This follows since each of the m-th roots of unity has norm one, and we are adding $\phi(m)$ of them with coefficients bounded by $\|a\|$. Clearly, for any two elements $a, b \in \mathbb{Z}[X]/\Phi_m(X)$ we have $\|a+b\|^{can} \le \|a\|^{can} + \|b\|^{can}$, and since the primitive m-th roots of unity are all roots of $\Phi_m(X)$ then $\|ab \bmod \Phi_m(X)\|^{can} = \|ab\|^{can} \le \|a\|^{can} \cdot \|b\|^{can}$. Similarly for n-vectors $\mathbf{a}, \mathbf{b} \in (\mathbb{Q}[X]/\Phi_m(X))^n$ we get $\| \langle \mathbf{a}, \mathbf{b} \rangle \bmod \Phi_m(X)\|^{can} \le n \cdot \|\mathbf{a}\|^{can} \cdot \|\mathbf{b}\|^{can}$.

Also, for every m there exists a "ring constant" c_m (which is a real number) such that for all $a \in \mathbb{Z}[X]/\Phi_m(X)$ it holds that $\|a\| \le c_m \cdot \|a\|^{can}$; see [6] for a discussion of c_m. Another property of the canonical embedding norm that we use below, is that a nonzero integer polynomial must have norm at least one:

Lemma 10. *Let $a \in \mathbb{Z}[X]/\Phi_m(X)$ for some integer m, then $\|a\|^{can} \ge 1$.*

Proof. Since a is a nonzero integer polynomial, then the result of the complex product $\prod_{k \in \mathbb{Z}_m^*} a(\rho^k)$ must be a nonzero integer, and therefore it has magnitude at least 1. It follows that some of the terms in the product must have magnitude 1 or more, hence the l_∞ norm of $\mathcal{E}(a)$ is at least 1. □

Modular Reduction in Canonical Embedding. To talk about the canonical norm of elements in $\mathbb{Z}_q[X]/\Phi_m(X)$ (i.e., polynomials reduced both mod $\Phi_m(X)$ and mod q), we define the "canonical embedding norm reduced mod q", denoted $|a|_q^{can}$, as the smallest norm $\|b\|^{can}$ among all the polynomials that are congruent to a modulo q. Namely, for $a \in \mathbb{Z}[X]/\Phi_m(X)$ we denote

$$|a|_q^{can} \stackrel{\text{def}}{=} \min\{ \ \|b\|^{can} : \ b \in \mathbb{Z}[X]/\Phi_m(X), \ b \equiv a \pmod q \ \}.$$

(We note that the minimum exists, even though we take it over an infinite set, since the set $\{\mathcal{E}(b) : b \equiv a \pmod q\}$ is a coset of a lattice.) Sometimes we may want to talk about the specific polynomial where the minimum is

obtained, namely the polynomial b satisfying $b \equiv a \pmod q$ and $\|b\|^{can} = |a|_q^{can}$. If this polynomial is unique, then we call it the "canonical reduction mod q of a" and denote it by

$$\overset{\text{can}}{[a]}_q \stackrel{\text{def}}{=} \operatorname{argmin}\{ \|b\|^{can} : b \in \mathbb{Z}[X]/\Phi_m(X),\ b \equiv a \pmod q \}.$$

We stress that our cryptosystem never needs to compute the canonical embedding (or the canonical reduction, or the canonical norm) of polynomials, it is only in the analysis of this scheme that we use these terms.

Obviously, for any element $a \in \mathbb{Z}[X]/\Phi_m(X)$ and any modulus q, the reduced canonical embedding norm is not more than the canonical embedding norm, namely $|a|_p^{can} \le \|a\|^{can}$. Similarly, it is easy to check that if $c \equiv ab \bmod (\Phi_m(X), q)$ then $|c|_q^{can} \le |a|_q^{can} \cdot |b|_q^{can}$. A corollary of Lemma 10 (that we use in our analysis of modulus switching) is that an element with small enough canonical embedding norm must be the unique canonical reduction mod q of its coset:

Lemma 11. *Let m, q be integers, and let $a \in \mathbb{Z}[X]/\Phi(m)$ be such that $\|a\|^{can} < q/2$. Then for any $b \in \mathbb{Z}[X]/\Phi_m$ such that $b \neq a$ but $b \equiv a \bmod q$, it holds that $\|b\|^{can} \gneq \|a\|^{can} = |b|_q^{can}$. Hence for all $b \equiv a \bmod q$ we have $a = \overset{\text{can}}{[b]}_q$.*

Proof. Fix any $b \in \mathbb{Z}[X]/\Phi_m$ such that $b \neq a$ but $b \equiv a \bmod q$. Then $\frac{b-a}{q}$ is a nonzero integer polynomial, and by Lemma 10 its canonical embedding has an entry of magnitude ≥ 1. This implies that $\mathcal{E}(b)$ has an entry of distance at least q from the corresponding entry in $\mathcal{E}(a)$. Since that entry in $\mathcal{E}(a)$ has magnitude $< q/2$, then the one in $\mathcal{E}(b)$ must have magnitude $> q/2$, and therefore $\|b\|^{can} > q/2 > \|a\|^{can}$. It follows that a has the unique smallest canonical embedding norm among all the polynomials in its coset mod q. □

D.2 Our Cryptosystem

In terms of operations, our cryptosystem is almost identical to the BGV cryptosystem [3], where all the operations are done modulo $\Phi_m(X)$. However, our analysis of (the functionality of) this cryptosystem is somewhat different, in that we keep track of the canonical norm of "the noise" rather than the norm of its coefficient vector. Specifically, we maintain the invariant that if $\mathbf{c}$ is a ciphertext encrypting the aggregate plaintext $a \in \mathbb{Z}_p[X]/\Phi_m(X)$ relative to secret key $\mathbf{s}$ and modulus q, then in the ring $\mathbb{Z}_q[X]/\Phi_m(X)$ we have the equality

$$\langle \mathbf{c}, \mathbf{s} \rangle \ = \ p \cdot u \ + \ a \pmod{\Phi_m(X), q}, \tag{1}$$

where $u \in \mathbb{Z}[X]/\Phi_m(X)$ has small canonical norm mod q, $|u|_q^{can} \ll q$.

Decryption. We claim that as long as this invariant holds, we can use $\mathbf{s}$ to decrypt $\mathbf{c}$. This can be done in one of two ways:

- If the "ring constant" c_m happens to be small enough (i.e., much smaller than q), then from $\|u\|^{can} \ll q$ and $p \ll q$ and $c_m \ll q$ we conclude that also $\|p \cdot u\| \le c_m \cdot p \cdot \|u\|^{can} \ll q$, which means that the coefficient vector of the noise has small norm and decryption works just as in standard BGV cryptosystems. For example for prime values of m the constant c_m is equal to approximately $4/\pi$, [6].
- Otherwise, we "lift" decryption to work modulo $X^m - 1$ rather than modulo $\Phi_m(X)$, and use the fact that the "ring constant" of $\mathbb{Z}[X]/(X^m - 1)$ is small (namely, it is $\sqrt{m}$).

Describing the second option in more detail, Lemma 12 below tells us that there exists an integer polynomial $G \in \mathbb{Z}[X]/(X^m - 1)$ such that $G(\alpha) = m$ for every complex primitive m-th root of unity α, and $G(\beta) = 0$ for every complex non-primitive m-th root of unity β. This means in particular that $G \equiv m \pmod{\Phi_m(X)}$ (in words, the polynomial G reduces to the constant m modulo Φ_m).

Computing $b \leftarrow G \cdot \langle \mathbf{c}, \mathbf{s} \rangle \bmod (X^m - 1, q)$, we get $b = p \cdot Gu + Ga \pmod{X^m - 1, q}$, due to Equation (1). We now observe that the evaluation of the polynomial Gu in all the m-th roots of unity must be small: For the primitive roots this evaluation is only m times that of u (which is small by our invariant), and for the non-primitive roots this evaluation is zero (since G evaluates to zero in these roots). Therefore the canonical norm of Gu in $\mathbb{Z}[X]/(X^m - 1)$ is small and therefore also the norm of its coefficient vector is small, so it can be decrypted as in standard BGV cryptosystems. Namely, we have no wraparound so setting $b' \leftarrow b \bmod p$ we have $b' = Ga \in \mathbb{Z}[X](X^m - 1)$. If we now further reduce modulo $\Phi_m(X)$, $b'' \leftarrow b' \bmod \Phi_m$, we get $b = m \cdot a \in \mathbb{Z}[X]/\Phi_m(X)$ (because $G \equiv m \pmod{\Phi_m(X)}$). Finally we can multiply by $(m^{-1} \bmod p)$ to get $a = m^{-1} \cdot b'' \bmod p$.

Lemma 12. *For any integer m there is an integer polynomial G_m of degree $\leq m - 1$, such that $G_m(\alpha) = m$ for every complex primitive m-th root of unity α, and $G_m(\beta) = 0$ for every complex non-primitive m-th root of unity β. Moreover the Euclidean norm of G_m's coefficient vector is $\sqrt{m \cdot \phi(m)}$.*

Proof. Clearly there exists a complex polynomial of degree $\leq m - 1$ which evaluates to m in the primitive m-th roots of unity and to zero in the non-primitive m-th roots of unity. We only need to show that this polynomial has integer coefficients, and that it has a low-norm coefficient vector.

To show that, let D be the $m \times m$ DFT matrix (i.e., the Vandemonde matrix on complex m-th roots of unity, $D_{ij} = \rho^{ij}$ for some fixed primitive m-th root of unity ρ). Denote the coefficient vector of G by $\boldsymbol{g}$, and the vector of values that it assumes in all the m-th roots of unity by $\boldsymbol{v}$ (so $\boldsymbol{v}$ is a vector of m's and 0's), and we have $\boldsymbol{v} = D\boldsymbol{g}$. Recalling that the inverse of D is $D^{-1} = D^*/m$ (with D^* the conjugate transpose of D), and considering the 0-1 vector $\boldsymbol{v'} = \boldsymbol{v}/m$, we have that $\boldsymbol{g} = D^*\boldsymbol{v'}$. Each coefficient in G is therefore a 0-1 combination of the entries in one row of D^*, with the 1's in the positions corresponding to the primitive roots of unity. Specifically, the coefficient of x^j in G is $g_j = \sum_i (\rho^{-j})^i$, where the sum goes over all indexes $i \in \mathbb{Z}_m^*$. Since the sum is symmetric over the primitive roots of unity, then it must sum to an integer. Hence G must be an integer polynomial.

Finally, recall that the matrix D^* is orthogonal with rows of norm $\sqrt{m}$, hence the l_2 norm of $\boldsymbol{g}$ is $\sqrt{m}$ times the l_2-norm of $\boldsymbol{v'}$. Since the number of 1's in $\boldsymbol{v'}$ is exactly $\phi(m)$, then the l_2 norm of $\boldsymbol{v'}$ is $\sqrt{\phi(m)}$, and therefore the l_2 norm of $\boldsymbol{g}$ is $\sqrt{m\phi(m)}$. □

Having described decryption, we now proceed to describe all the other elements of our cryptosystem, namely key-generation, encryption, addition, "raw multiplication", key-switching, modulus switching, and Galois group actions. All these components (bar the last) are very similar to their counterpart in the BGV cryptosystem [3], but their analysis is slightly different.

Key Generation. The parameters of the scheme include the integer m (that defines the polynomial Φ_m), the integer p (that defines the aggregate plaintext space $\mathbb{Z}_p[X]/\Phi_m$), and the sequence of moduli $q_0 > q_1 > \cdots > q_L$.

Key generation is as in the ring-LWE-based version BGV [3] over the ring $\mathbb{Z}[x]/\Phi_m$. That is, for appropriate $N = \text{polylog}(q_0, m)$, one chooses $\mathfrak{s}_0, \epsilon_{0,1}, \ldots \epsilon_{0,N} \in \mathbb{Z}[X]/\Phi_m$ (with l_∞ coefficient norm $\ll q_0$) as well as a random elements $\alpha_{0,1}, \ldots, \alpha_{0,N} \in_R \mathbb{Z}_{q_0}[X]/\Phi_m$, and computes $\beta_{0,i} \leftarrow \alpha_{0,i}\mathfrak{s}_0 + p \cdot \epsilon_{0,i} \bmod (\Phi_m(X), q_0)$. The level-0 secret key is $\mathbf{s}_0 = [1, \mathfrak{s}_0]$, and the corresponding public encryption key includes the vectors $\mathbf{b}_i = [\beta_{0,i}, -\alpha_{0,i}]$.

In addition to these keys, the key-generation procedure chooses other secret key vectors for the other levels, and generates the key-switching matrices between them, as described in Section D.2 below.

Encryption. Encryption is as in BGV. An aggregate plaintext $a \in \mathbb{Z}_p[X]/\Phi_m(X)$ is encrypted by choosing random short elements $\tau_1, \ldots \tau_N \in \mathbb{Z}[X]/\Phi_m$ (with l_∞ coefficient norm $\ll q_0$) and setting

$$\mathbf{c} = [c_0, c_1] \leftarrow [a, 0] + \sum_{i=1}^{N} \tau_i \cdot \mathbf{b}_i \bmod (\Phi_m(X), q_0). \tag{2}$$

(Actually, the τ_i's can be chosen as elements of $\mathbb{Z}[x]/\Phi_m$ with 0/1 coefficients, versus merely being short.)

It is easy to show that semantic security reduces to the hardness of the decision ring-LWE problem for the ring $\mathbb{Z}_q[X]/\Phi_m$ and the distributions used to sample the short elements.

To see that our invariant holds with respect to the level-0 secret key $\mathbf{s}_0$ and freshly encrypted ciphertexts, note that Equation (2) implies that $\mathbf{c} = [a, 0] + \sum_{i=1}^{N} \tau_i \cdot \mathbf{b}_i \pmod{\Phi_m(X), q_0}$, and therefore

$$\begin{aligned}\langle \mathbf{c}, \mathbf{s}_0 \rangle = a + \sum_{i=1}^{N} \tau_i \langle \mathbf{s}_0, \mathbf{b}_i \rangle &= a + p \cdot \sum_{i=1}^{N} \tau_i \cdot \epsilon_i \\ &= a + p \cdot \sum_{i=1}^{N} \tau_i \cdot \epsilon_i \pmod{\Phi_m(X), q_0}\end{aligned}$$

and the since all the τ_i's and ϵ_i's are small (and therefore also have small canonical embedding norm), then the canonical embedding norm of the polynomial $u = \sum_{i=1}^{N} \tau_i \cdot \epsilon_i \bmod (\Phi_m(X), q_0)$ is small.

Addition. Adding two ciphertext vectors that are defined with respect to the same secret key and modulus is just standard addition in $\mathbb{Z}_q[X]/\Phi_m(X)$. Clearly, if $\langle \mathbf{c}, \mathbf{s} \rangle = p \cdot u + a$ and $\langle \mathbf{c}', \mathbf{s} \rangle = p \cdot u' + a'$ then also $\langle \mathbf{c} + \mathbf{c}', \mathbf{s} \rangle = p \cdot (u + u') + (a + a')$, and the canonical embedding norm of $u + u'$ is still small.

"Raw Multiplication". As in the BV/BGV family of cryptosystems [5, 4, 3], "raw multiplication" of two ciphertext vectors (defined with respect to the same modulus) is done using tensor product. Namely, if we have ciphertext vector $\mathbf{c}$ which is decrypted to a under $\mathbf{s}$ and q, and another vector $\mathbf{c}'$ which is decrypted to a' under $\mathbf{s}'$ and q, then we set $\tilde{\mathbf{c}} = \text{vector}(\mathbf{c} \otimes \mathbf{c}') \bmod (\Phi_m(X), q)$ (where vector$(\cdot)$ opens the matrix into a vector using some appropriate ordering). Denoting $\tilde{\mathbf{s}} = \text{vector}(\mathbf{s} \otimes \mathbf{s}') \bmod (\Phi_m(X), q)$, we thus have

$$\begin{aligned}\langle \tilde{\mathbf{c}}, \tilde{\mathbf{s}} \rangle = \mathbf{s}^t (\mathbf{c} \otimes \mathbf{c}') \mathbf{s}' &= \langle \mathbf{c}, \mathbf{s} \rangle \cdot \langle \mathbf{c}', \mathbf{s}' \rangle \\ &= (p \cdot u + a) \cdot (p \cdot u' + a') = p \cdot (puu' + ua' + au') + aa' \pmod{\Phi_m(X), q}.\end{aligned}$$

Since the canonical embedding norm of $\tilde{u} = puu' + ua' + au' \bmod (\Phi_m(X), q)$ is still small, it means that $\tilde{\mathbf{c}}$ is a valid ciphertext with respect to $\tilde{\mathbf{s}}$ and q, which is decrypted to aa'.

Key Switching. A crucial component of the BV/BGV cryptosystems is the ability to translate a ciphertext with respect to one secret key into a ciphertext that decrypts to the same thing under another secret key. This is used, for example, to translate the "extended ciphertext" that we get from raw multiplication back to a normal ciphertext, or to translate two ciphertext vectors with respect to different keys into ciphertexts with respect to the same key, so that they can be added or raw-multiplied.

Let $\mathbf{s}$ be a secret-key vector over $\mathbb{Z}_q[X]/\Phi_m(X)$, and consider another 2-element secret-key vector $\mathbf{t} \in (\mathbb{Z}_q[X]/\Phi_m(X))^2$ whose first entry is 1. To allow translation from $\mathbf{s}$-ciphertexts to $\mathbf{t}$-ciphertexts, we first encode $\mathbf{s}$ in a redundant manner by computing $2^i \mathbf{s} \bmod q$ for $i = 0, 1, \ldots, l = \lceil \log q \rceil$ and concatenating all these

vectors to form

$$\hat{\mathbf{s}} \;=\; \mathsf{Powersof2}_q(\mathbf{s}) \;\stackrel{\text{def}}{=}\; [\mathbf{s} \mid 2\mathbf{s} \mid 4\mathbf{s} \mid \;\ldots\; \mid 2^l\mathbf{s}] \bmod q.$$

Then we choose a random low coefficient norm vector $\mathbf{v}$ over $\mathbb{Z}_q[X]/\Phi_m(X)$ of the same dimension as $\hat{\mathbf{s}}$ (call this dimension d), and a matrix $R \in (\mathbb{Z}_q[X]/\phi_m)^{2\times d}$ which is chosen at random from the orthogonal space to $\mathbf{t}$, namely $\mathbf{t}R = \mathbf{0} \pmod{\Phi_m(X), q}$. The key-switching matrix from $\mathbf{s}$ to $\mathbf{t}$ is then set as

$$W = W[\mathbf{s} \to \mathbf{t}] = \begin{bmatrix} \hat{\mathbf{s}} + p\mathbf{v} \\ -\;0\;- \end{bmatrix} \;+\; R \mod (\Phi_m(X), q)$$

Again it is easy to show that if decision ring-LWE is hard for the ring $\mathbb{Z}_q[X]/\Phi_m(X)$ and the distributions used to sample $\mathbf{t}$ and $\mathbf{v}$, then the matrix W above is pseudo-random, even for someone who knows $\mathbf{s}$.

Given a ciphertext vector $\mathbf{c}$ (over $\mathbb{Z}_q[X]/\Phi_m(X)$) that satisfies our invariant with respect to $\mathbf{s}$ and q, we use W to translate it into another vector $\mathbf{c}'$ that satisfies our invariant with respect to $\mathbf{t}$ and q, as follows: First, for $i = 0, 1, \ldots, l = \lceil \log q \rceil$ we denote by $\mathbf{c}_i$ the vector over $\mathbb{Z}_2[X]/\Phi_m(X)$ containing the i'th bits from all the coefficients of all the entries of $\mathbf{c}$. Namely:

$$\mathbf{c}_0 = \mathbf{c} \bmod 2, \quad \text{and } \mathbf{c}_i = 2^{-i} \cdot \big((\mathbf{c} \bmod 2^{i+1}) - \sum_{j<i} 2^j \mathbf{c}_j\big) \text{ for } i > 0.$$

Then the bit-decomposition of $\mathbf{c}$ is the concatenation of all these vectors,

$$\hat{\mathbf{c}} = \mathsf{BitDecomp}(\mathbf{c}) \;\stackrel{\text{def}}{=}\; [\mathbf{c}_0 \mid \mathbf{c}_1 \mid \;\ldots\; \mid \mathbf{c}_l].$$

Clearly $\hat{\mathbf{c}}$ has low norm coefficient vectors, since they are all 0-1 vectors, and we have $\langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle = \langle \mathbf{c}, \mathbf{s} \rangle$ over $\mathbb{Z}_q[X]$ (and therefore also over $\mathbb{Z}_q[X]/\Phi_m(X)$). Switching keys from $\mathbf{s}$ to $\mathbf{t}$ is done simply by setting $\mathbf{c}' \leftarrow W\hat{\mathbf{c}} \bmod (\Phi_m(X), q)$. To see that this maintains our invariant, assume that for some $a \in \mathbb{Z}_p[X]/\Phi_m(X)$ we have $\langle \mathbf{c}, \mathbf{s} \rangle = p \cdot u + a \pmod{\Phi_m(X), q}$, where u has low canonical embedding norm. Then:

$$\begin{aligned} \langle \mathbf{c}', \mathbf{t} \rangle = \mathbf{t}W\hat{\mathbf{c}} \;=\; \mathbf{t} \begin{bmatrix} \hat{s} + p\mathbf{v} \\ -\;0\;- \end{bmatrix} \hat{\mathbf{c}} \;\stackrel{(a)}{=}\; \langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle + p \cdot \langle \hat{\mathbf{c}}, \mathbf{v} \rangle \\ \stackrel{(b)}{=}\; \langle \mathbf{c}, \mathbf{s} \rangle + p \cdot \langle \hat{\mathbf{c}}, \mathbf{v} \rangle \;=\; p \cdot \underbrace{(u + \langle \hat{\mathbf{c}}, \mathbf{v} \rangle)}_{u'} + a \pmod{\Phi_m(X), q}, \end{aligned}$$

where Equality (a) holds since the first entry of $\mathbf{t}$ is 1, and Equality (b) follows from $\langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle = \langle \mathbf{c}, \mathbf{s} \rangle$. Finally, since both $\mathbf{v}$ and $\hat{\mathbf{c}}$ have low canonical embedding norm (because they have low coefficient norm), then so has $\langle \hat{\mathbf{c}}, \mathbf{v} \rangle$ and therefore also $u' = \langle \hat{\mathbf{c}}, \mathbf{v} \rangle + u \bmod (\Phi_m(X), q)$.

Galois Group Actions. Recall that a Galois group action is obtained by applying the transformation $f(X) \mapsto f(X^i) \bmod (\Phi_m(X), q)$ for some $i \in \mathbb{Z}_m^*$ to all the polynomials in our ciphertext vectors, secret keys, etc. Assume that we have $\langle \mathbf{c}, \mathbf{s} \rangle = p \cdot u + a \pmod{\Phi_m(X), q}$, and define $\mathbf{c}^{(i)}, \mathbf{s}^{(i)}, u^{(i)}, a^{(i)}$ as what you get by applying the above Galois group action to $\mathbf{c}, \mathbf{s}, u, a$, respectively. Our invariant means that for some polynomial $k \in \mathbb{Z}_q[X]$ we have

$$\sum_j \mathbf{c}_j(X)\mathbf{s}_j(X) = p \cdot u(X) + a(X) + k(X)\Phi_m(X) \;\text{ (equality in } \mathbb{Z}_q[X]), \tag{3}$$

and therefore also for every i

$$\sum_j \mathbf{c}_j(X^i)\mathbf{s}_j(X^i) = p \cdot u(X^i) + a(X^i) + k(X^i)\Phi_m(X^i) \;\text{ (equality in } \mathbb{Z}_q[X]). \tag{4}$$

Equation (4) follows since the two sides of Equation (3) are identical as formal polynomials over $\mathbb{Z}_q$, and therefore they must coincide also as functions over any characteristic-q field. It follows that the functions on both sides of Equation (4) must also coincide over any characteristic-q field, and therefore the two sides must be identical as formal polynomials over $\mathbb{Z}_q$.

Recalling that if $i \in \mathbb{Z}_m^*$ then $\Phi(X)$ divides $\Phi(X^i)$, we obtain

$$\left\langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \right\rangle = \sum_j \mathbf{c}_j(X^i)\mathbf{s}_j(X^i) = p \cdot u(X^i) + a(X^i) = p \cdot u^{(i)} + a^{(i)} \pmod{\Phi_m(X), q},$$

as needed. Observing that for $i \in \mathbb{Z}_m^*$ the canonical embeddings of u and $u^{(i)}$ are just a permutation of each other (and hence have the same norm) we deduce that our invariant in maintained under the transformation $X \mapsto X^i$ whenever $i \in \mathbb{Z}_m^*$.

Modulus Switching. Our modulus switching procedure works exactly as in the BGV cryptosystem. Namely, to switch a ciphertext $\mathbf{c}$ (in coefficient representation) from q_i to q_{i+1}, we just scale the coefficient vectors in $\mathbf{c}$ by a q_{i+1}/q_i factor, and then round the result to get an integer polynomial vector $\mathbf{c}'$ such that $\mathbf{c}' \equiv \mathbf{c} \pmod{p}$.

Definition 3 (Scale)**.** *For a vector $\mathbf{c}$ over $\mathbb{Z}[X]/\Phi_m(X)$ and integers $q_i > q_{i+1} > p$, define $\mathbf{c}' \leftarrow \mathsf{Scale}(\mathbf{c}, q_i, q_{i+1}, p)$ to be the vector over $\mathbb{Z}[X]/\Phi_m(X)$ closest to $(p/q)\cdot\mathbf{c}$ (in coefficient representation) that satisfies $\mathbf{c}' \equiv \mathbf{c} \pmod{p}$.*

Our analysis, however, is a little different than in [3]. The proof from [3, Lemma 4] relies on the fact that the coefficient vector of $[\langle \mathbf{c}, \mathbf{s}\rangle]_{q_i}$ has low norm, whereas in out case we instead have that this polynomials has low canonical embedding norm mod q_i. We therefore re-prove this lemma under our new condition.

Lemma 13. *Let $q_i > q_{i+1} > p$ be positive integers satisfying $q_i = q_{i+1} = 1 \pmod{p}$. Let $\mathbf{c}, \mathbf{s}$ be two n-vectors over $\mathbb{Z}[X]/\Phi_m(X)$ such that $|\langle \mathbf{c}, \mathbf{s}\rangle|_{q_i}^{can} < q_i/2 - \frac{q_i}{q_{i+1}} \cdot pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can}$, and let $\mathbf{c}' = \mathsf{Scale}(\mathbf{c}, q_i, q_{i+1}, p)$. Denoting $e = \langle \mathbf{c}, \mathbf{s}\rangle \bmod \Phi_m(X)$ and $e' = \langle \mathbf{c}', \mathbf{s}\rangle \bmod \Phi_m(X)$ (arithmetic in $\mathbb{Z}[X]/\Phi_m(X)$), it holds that*

$$\begin{aligned} \overset{\text{can}}{[e']}_{q_{i+1}} &\equiv \overset{\text{can}}{[e]}_{q_i} \pmod{p} \text{ (in coefficient representation), and} \\ |e'|_{q_{i+1}}^{can} &< \frac{q_{i+1}}{q_i} \cdot |e|_{q_i}^{can} + pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can} \end{aligned}$$

Proof. For some $k \in \mathbb{Z}[X]/\Phi_m(X)$, we have $\overset{\text{can}}{[e]}_{q_i} = \langle \mathbf{c}, \mathbf{s}\rangle - q_i k$, where the equality is over $\mathbb{Z}[X]/\Phi_m(X)$. For the same k, let $e'' = e' - q_{i+1}k \in \mathbb{Z}[X]/\Phi_m(X)$. Since $\mathbf{c}' \equiv \mathbf{c} \pmod{p}$ and $q_i \equiv q_{i+1} \pmod{p}$, then also

$$e'' = \langle \mathbf{c}', \mathbf{s}\rangle - q_{i+1}k \equiv \langle \mathbf{c}, \mathbf{s}\rangle - q_i k = \overset{\text{can}}{[e]}_{q_i} \pmod{\Phi_m(X), p}.$$

It therefore suffices to prove that $e'' = \overset{\text{can}}{[e']}_{q_{i+1}}$ (equality over $\mathbb{Z}[X]/\Phi_m(X)$) and that it has small enough norm.

Denote the distance between $\frac{q_{i+1}}{q_i} \cdot \mathbf{c}$ and its rounded version $\mathbf{c}'$ by $\delta \stackrel{\text{def}}{=} \mathbf{c}' - \frac{q_{i+1}}{q_i}\mathbf{c}$. Then δ is a vector over $\mathbb{Q}[X]/\Phi_m(X)$, and the coefficient-vectors in δ all have entries in $[-p/2, p/2)$. Moreover, we have

$$\begin{aligned} e'' = \langle \mathbf{c}', \mathbf{s}\rangle - q_{i+1}k &= \frac{q_{i+1}}{q_i}\langle \mathbf{c}, \mathbf{s}\rangle + \langle \delta, \mathbf{s}\rangle - q_{i+1}k \\ &= \frac{q_{i+1}}{q_i}\big(\langle \mathbf{c}, \mathbf{s}\rangle - q_i k\big) + \langle \delta, \mathbf{s}\rangle = \frac{q_{i+1}}{q_i} \cdot \overset{\text{can}}{[e]}_{q_i} + \langle \delta, \mathbf{s}\rangle . \end{aligned} \tag{5}$$

Considering the polynomial $\langle \delta, \mathbf{s} \rangle \in \mathbb{Q}[X]/\Phi_m(X)$, we can bound its canonical embedding norm by:

$$\| \langle \delta, \mathbf{s} \rangle \|^{can} \;\leq\; n \cdot \|\delta\|^{can} \cdot \|\mathbf{s}\|^{can} \;\leq\; n \cdot \phi(m) \cdot \|\delta\| \cdot \|\mathbf{s}\|^{can} \;\leq\; pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can}.$$

From Equation (5) we now get:

$$\begin{aligned} \|e''\|^{can} \;&\leq\; \frac{q_{i+1}}{q_i} \cdot |e|^{can}_{q_i} + \| \langle \delta, \mathbf{s} \rangle \|^{can} \;\leq\; \frac{q_{i+1}}{q_i} \cdot |e|^{can}_{q_i} + pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can} \\ &<\; \left(\frac{q_{i+1}}{2} - pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can}\right) + pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can} \;=\; \frac{q_{i+1}}{2} \end{aligned} \tag{6}$$

Finally, Lemma 11 implies that $e'' = \overset{\text{can}}{[e']}_{q_{i+1}}$, completing the proof. □

It follows immediately from Lemma 13 that if $\mathbf{c}$ satisfies our invariant with respect to $\mathbf{s}$ and q_i, and if the canonical embedding norm of $\mathbf{s}$ is small enough so that we have $| \langle \mathbf{c}, \mathbf{s} \rangle |^{can}_{q_i} < q_i/2 - \frac{q_i}{q_{i+1}} \cdot pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can}$, then the scaled vector $\mathbf{c}' = \mathsf{Scale}(\mathbf{c}, q_i, q_{i+1}, p)$ satisfies our invariant with respect to the same $\mathbf{s}$ and the new modulus q_{i+1}.

Variants. We note that one can optimize BGV key generation and encryption using a cute trick by Brakerski and Vaikuntanathan [5] (following [15]). This reduces the public key size and encryption time, without changing the scheme in an any way that affects the applicability of our techniques; we still obtain FHE with polylog overhead using BGV with BV's optimizations. (We note that our techniques can be applied to the cryptosystem of BV [5] as well, but one needs to use BGV's noise management technique to reduce the overhead to polylog.)

In BV key generation [5], for level-0, one only needs to choose low-norm elements $\mathfrak{s}_0, \epsilon_0 \in \mathbb{Z}[X]/\Phi_m(X)$ (with coefficient norm $\ll q_L$) as well as a random element $\alpha_0 \in_R \mathbb{Z}_{q_0}[X]/\Phi_m(X)$, and computing $\beta_0 \leftarrow -\alpha_0\mathfrak{s}_0 + p \cdot \epsilon_0 \bmod (\Phi_m(X), q_0)$. The level-0 secret key is $\mathbf{s}_0 = [1, \mathfrak{s}_0]$, and the corresponding public encryption key is $\mathbf{b} = [\beta_0, \alpha_0]$. This approach reduces level-0 key size by factor of $O(\log q_0)$. One generates keys for the other levels similarly.

In BV encryption, an aggregate plaintext $a \in \mathbb{Z}_p[X]/\Phi_m(X)$ is encrypted by choosing three random short elements $\tau, \epsilon_1, \epsilon_2 \in \mathbb{Z}_{q_0}[X]/\Phi_m(X)$ and setting

$$\mathbf{c} = [c_0, c_1] \leftarrow [\tau\beta_0, \tau\alpha_0] + p \cdot [\epsilon_1, \epsilon_2] + [a, 0] \bmod (\Phi_m(X), q_0). \tag{7}$$

It is easy to show that semantic security reduces to the hardness of the decision ring-LWE problem for the ring $\mathbb{Z}_q[X]/\Phi_m(X)$ and the distributions used to sample $\mathfrak{s}_0, \tau$, and $\epsilon, \epsilon_1, \epsilon_2$.

To see that our invariant holds with respect to the level-0 secret key $\mathbf{s}_0$ and freshly encrypted ciphertexts, note that Equation (7) implies that $\mathbf{c} = [\tau\beta_0, \tau\alpha_0] + p \cdot [\epsilon_1, \epsilon_2] + [a, 0] \pmod{\Phi_m(X), q_0}$, and therefore

$$\begin{aligned} \langle \mathbf{c}, \mathbf{s}_0 \rangle &= \tau\beta_0 + p\epsilon_1 + a + \mathfrak{s}(\tau\alpha_0 + p\epsilon_2) \;=\; -\tau\mathfrak{s}\alpha_0 + p\tau\epsilon_0 + p\epsilon_1 + a + \mathfrak{s}(\tau\alpha_0 + p\epsilon_2) \\ &= p \cdot (\tau\epsilon_0 + \epsilon_1 + \mathfrak{s}\epsilon_2) + a \pmod{\Phi_m(X), q_0} \end{aligned}$$

and the polynomial $u = (\tau\epsilon_0 + \epsilon_1 + \mathfrak{s}\epsilon_2) \bmod (X^m - 1, q_0)$ has low coefficient norm, and therefore also low canonical embedding norm. When using BV encryption and key generation, the other aspects of the scheme remain the same.

E A Delayed-Reduction Technique

We describe here another variant, where we work with polynomials modulo $X^m - 1$ rather than polynomials modulo Φ_m, and reduce back mod Φ_m only upon decryption. Importantly, we still want to base our security on the hardness of ring-LWE with respect to the ring $\mathbb{Z}_q[X]/\Phi_m(X)$ (recall that decision ring-LWE is easy modulo $X^m - 1$, since it can be reduced to the one-dimensional problem modulo $X - 1$).

We can use Lemma 12 to "lift" the mod-$\Phi_m(x)$ polynomials in the cryptosystem into mod-$(X^m - 1)$ polynomials, simply by multiplying by the polynomial $G(X)$ from that lemma. (This has the effect of introducing an extra multiplicative factor of m, which we can correct upon decryption.) Note that since $G = 0 \pmod{\frac{X^m-1}{\Phi_m(x)}}$, then we can write $G(X) = \frac{X^m-1}{\Phi_m(x)} \cdot \mu(X)$ (equality over $\mathbb{Z}[X]$) for some integer polynomial μ. It follows that if we have two polynomials satisfying $u = v \pmod{\Phi_m}$ then $Gu = Gv \pmod{X^m - 1}$. This is because over $\mathbb{Z}[X]/(X^m - 1)$ we have $u = v + \tau\Phi_m$ for some integer polynomial τ, and so

$$Gu \;=\; G(v + \tau\Phi_m) \;=\; Gu + (\frac{X^m - 1}{\Phi_m}\mu) \cdot \tau\Phi_m \;=\; Gu + (X^m - 1) \cdot \mu\tau \;=\; Gu \pmod{X^m - 1}$$

In our variant of the BGV cryptosystem, ciphertexts are vectors over the ring $\mathbb{Z}[X]/(X^m - 1)$, secret keys are vectors over the sub-ring $\mathbb{Z}[X]/\Phi_m$, and aggregate plaintexts are elements in $\mathbb{Z}_p[X]/\Phi_m$. We maintain the invariant that if $\mathbf{c}$ is a ciphertext encrypting the aggregate plaintext a relative to secret key $\mathbf{s}$ and modulus q, then in the ring $\mathbb{Z}_q[X]/(X^m - 1)$ we have the equality

$$G \cdot \langle \mathbf{c}, \mathbf{s} \rangle \;=\; p \cdot G \cdot u \;+\; G \cdot a \pmod{X^m - 1, q}, \tag{8}$$

where $u \in \mathbb{Z}[X]/(X^m - 1)$ has coefficient vector with small l_2-norm, $\|u\|_2 \ll q$. Note that we can use $\mathbf{s}$ to decrypt $\mathbf{c}$ by setting $b \leftarrow G \cdot \langle \mathbf{c}, \mathbf{s} \rangle \bmod (X^m - 1, q)$, then recovering $a = m^{-1} \cdot b \bmod (\Phi_m, p)$. Since both b and $p \cdot Gu + Ga \pmod{X^m - 1}$ have coefficients smaller than $q/2$ in absolute value, then we have the equality $b = p \cdot Gu + Ga$ holding over $\mathbb{Z}[X]/(X^m - 1)$, without reduction modulo q. We thus have $b = Ga \pmod{X^m - 1, p}$, so also $b = Ga = m \cdot a \pmod{\Phi_m, p}$, so indeed $a = b \cdot m^{-1} \pmod{\Phi_m, p}$.

Having described decryption, we now proceed to describe all the other elements of our cryptosystem, namely key-generation, encryption, addition, "raw multiplication", key-switching, modulus switching, and Galois group actions. All these components (bar the last) are very similar to their counterpart in the BGV cryptosystem [3], except that we use some mix of mod-Φ_m and mod-$(X^m - 1)$ arithmetic, using multiplication-by-G and Equation (8) to move between them.

E.1 Key generation

The parameters of the scheme include the integer m (that defines the polynomials Φ_m and $X^m - 1$), the integer p (that defines the aggregate plaintext space $\mathbb{Z}_p[X]/\Phi_m$), and the sequence of moduli $q_0 > q_1 > \cdots > q_L$.

Key generation is as in the ring-LWE-based version BGV [3] over the ring $\mathbb{Z}[x]/\Phi_m$. That is, for appropriate $N = \text{polylog}(q_0, m)$, one chooses low-norm elements $\mathfrak{s}_0, \epsilon_{0,1}, \ldots \epsilon_{0,N} \in \mathbb{Z}[X]/\Phi_m$ (with l_2 norm $\ll q_0$) as well as a random elements $\alpha_{0,1}, \ldots, \alpha_{0,N} \in_R \mathbb{Z}_{q_0}[X]/\Phi_m$, and computes $\beta_{0,i} \leftarrow \alpha_{0,i}\mathfrak{s}_0 + p \cdot \epsilon_{0,i} \bmod (\Phi_m, q_0)$. The level-0 secret key is $\mathbf{s}_0 = [1, \mathfrak{s}_0]$, and the corresponding public encryption key includes the vectors $\mathbf{b}_i = [\beta_{0,i}, -\alpha_{0,i}]$.

In addition to these keys, the key-generation procedure chooses other secret key vectors for the other levels, and generates the key-switching matrices between them, as described in Section E.5 below.

E.2 Encryption

Encryption is as in BGV. An aggregate plaintext $a \in \mathbb{Z}_p[X]/\Phi_m(X)$ is encrypted by choosing random short elements $\tau_1, \ldots \tau_N \in \mathbb{Z}[X]/\Phi_m$ and setting

$$\mathbf{c} = [c_0, c_1] \leftarrow [a, 0] + \sum_{i=1}^{N} \tau_i \cdot \mathbf{b}_i \bmod (\Phi_m, q_0). \tag{9}$$

(Actually, the τ_i's can be chosen as elements of $\mathbb{Z}[x]/\Phi_m$ with 0/1 coefficients, versus merely being short.)

Note that freshly encrypted ciphertexts are vectors over the sub-ring $\mathbb{Z}[X]/\Phi_m(X)$, but later we allow evaluated ciphertexts to be in the larger ring $\mathbb{Z}[X]/(X^m - 1)$. It is easy to show that semantic security reduces to the hardness of the decision ring-LWE problem for the ring $\mathbb{Z}_q[X]/\Phi_m$ and the distributions used to sample the short elements.

To see that our invariant holds with respect to the level-0 secret key $\mathbf{s}_0$ and freshly encrypted ciphertexts, note that Equation (9) implies that $G \cdot \mathbf{c} = G([a, 0] + \sum_{i=1}^{N} \tau_i \cdot \mathbf{b}_i) \pmod{X^m - 1, q_0}$, and therefore

$$\begin{aligned} G \cdot \langle \mathbf{c}, \mathbf{s}_0 \rangle &= G(a + \sum_{i=1}^{N} \tau_i \langle \mathbf{s}_0, \mathbf{b}_i \rangle) \\ &= G(a + p \cdot \sum_{i=1}^{N} \tau_i \cdot \epsilon_i) \\ &= Ga + p \cdot G(\sum_{i=1}^{N} \tau_i \cdot \epsilon_i) \pmod{X^m - 1, q_0} \end{aligned}$$

and the coefficient vector of the polynomial $u = \sum_{i=1}^{N} \tau_i \cdot \epsilon_i \bmod (X^m - 1, q_0)$ has low l_2 norm.

We stress that the low l_2 norm of u depends crucially on our delayed reduction. Indeed, each of the polynomials $\{\tau_i\}, \{\epsilon_i\}, G$ has low l_2 norm, hence their products and sums over $\mathbb{Z}[X]$ would still have low norms. However, we do not know how to prove that the norm remains low when we reduce them modulo Φ_m, it is only because we reduce modulo $X^m - 1$ that we can argue that the norm remains low.

E.3 Addition

Adding two ciphertext vectors that are defined with respect to the same secret key and modulus is just standard addition in $\mathbb{Z}_q[X]/(X^m - 1)$. Indeed, if we have $G \cdot \langle \mathbf{c}, \mathbf{s} \rangle = p \cdot Gu + Ga$ and $G \cdot \langle \mathbf{c}', \mathbf{s} \rangle = p \cdot Gu' + Ga'$ (both over $\mathbb{Z}_q[X]/(X^m - 1)$) then also $G \cdot \langle \mathbf{c} + \mathbf{c}', \mathbf{s} \rangle = p \cdot G(u + u') + G(a + a')$, and the l_2 norm of the coefficient vector of $u + u'$ is still small.

E.4 "Raw multiplication"

As in the BV/BGV family of cryptosystems [5, 4, 3], "raw multiplication" of two ciphertext vectors (defined with respect to the same secret key and modulus) is done using tensor product. Namely, if we have ciphertext vector $\mathbf{c}$ which is decrypted to a under $\mathbf{s}$ and q, and another vector $\mathbf{c}'$ which is decrypted to a' under $\mathbf{s}$ and q, then we set $\tilde{\mathbf{c}} = \text{vector}(\mathbf{c} \otimes \mathbf{c}') \bmod (X^m - 1, q)$ (where vector$(\cdot)$ opens the matrix into a vector using some appropriate

ordering). Denoting $\tilde{\mathbf{s}} = \text{vector}(\mathbf{s} \otimes \mathbf{s}) \bmod (\Phi_m, q)$, we thus have

$$\begin{aligned} G \cdot \langle \tilde{\mathbf{c}}, \tilde{\mathbf{s}} \rangle &= G \cdot \mathbf{s}^t (\mathbf{c} \otimes \mathbf{c}') \mathbf{s} \;=\; G \cdot \langle \mathbf{c}, \mathbf{s} \rangle \cdot \langle \mathbf{c}', \mathbf{s} \rangle \\ &= (p \cdot Gu + Ga) \cdot \langle \mathbf{c}', \mathbf{s} \rangle \;=\; (p \cdot u + a) \cdot G \cdot \langle \mathbf{c}', \mathbf{s} \rangle \;=\; (p \cdot u + a) \cdot (p \cdot Gu' + Ga') \\ &= p \cdot G(puu' + ua' + au') + Gaa' \pmod{X^m - 1, q}. \end{aligned}$$

Since the coefficient vector of $\tilde{u} = puu' + ua' + au' \bmod (X^m - 1, q)$ still has small l_2 norm, it means that $\tilde{\mathbf{c}}$ is a valid ciphertext with respect to $\tilde{\mathbf{s}}$ and q, which is decrypted to aa'. Note that above we used mod-$(X^m - 1)$ arithmetic for the ciphertext and mod-Φ_m arithmetic for the secret key. This choice was made for convenience in other operations.

E.5 Key switching

A crucial component of the BV/BGV cryptosystems is the ability to translate a ciphertext with respect to one secret key into a ciphertext that decrypts to the same thing under another secret key. This is used, for example, to translate the "extended ciphertext" that we get from raw-multiplication back to a normal ciphertext, or to translate two ciphertext vectors with respect to different keys into ciphertexts with respect to the same key, so that they can be added or raw-multiplied.

Let $\mathbf{s}$ be a secret-key vector over $\mathbb{Z}_q[X]/\Phi_m$, and consider another 2-element secret-key vector $\mathbf{t} \in (\mathbb{Z}_q[X]/\Phi_m)^2$ whose first entry is 1. To allow translation from $\mathbf{s}$-ciphertexts to $\mathbf{t}$-ciphertexts, we first encode $\mathbf{s}$ in a redundant manner by computing $2^i\mathbf{s} \bmod q$ for $i = 0, 1, \ldots, l = \lceil \log q \rceil$ and concatenating all these vectors to form

$$\hat{\mathbf{s}} \;=\; \mathsf{Powersof2}_q(\mathbf{s}) \;\stackrel{\text{def}}{=}\; [\mathbf{s} \mid 2\mathbf{s} \mid 4\mathbf{s} \mid \ \ldots \ \mid 2^l\mathbf{s}] \bmod q.$$

Then we choose a random low l_2 norm vector $\mathbf{v}$ over $\mathbb{Z}_q[X]/\Phi_m$ of the same dimension as $\hat{\mathbf{s}}$ (call this dimension d), and a matrix $R \in (\mathbb{Z}_q[X]/\phi_m)^{2\times d}$ which is chosen at random from the orthogonal space to $\mathbf{t}$, namely $\mathbf{t}R = \mathbf{0} \pmod{\Phi_m, q}$. The key-switching matrix from $\mathbf{s}$ to $\mathbf{t}$ is then set as

$$W = W[\mathbf{s} \to \mathbf{t}] = \begin{bmatrix} \hat{\mathbf{s}} + p\mathbf{v} \\ -\ 0\ - \end{bmatrix} \;+\; R \mod (\Phi_m, q)$$

Again it is easy to show that if decision ring-LWE is hard for the ring $\mathbb{Z}_q[X]/\Phi_m(X)$ and the distributions used to sample $\mathbf{t}$ and $\mathbf{v}$, then the matrix W above is pseudo-random, even for someone who knows $\mathbf{s}$.

Given a ciphertext vector $\mathbf{c}$ (over $\mathbb{Z}_q[X]/(X^m - 1)$) that satisfies our invariant with respect to $\mathbf{s}$ and q, we use W to translate it into another vector $\mathbf{c}'$ that satisfies our invariant with respect to $\mathbf{t}$ and q, as follows: First, for $i = 0, 1, \ldots, l = \lceil \log q \rceil$ we denote by $\mathbf{c}_i$ the vector over $\mathbb{Z}_2[X]/(X^m - 1)$ containing the i'th bits from all the coefficients of all the entries of $\mathbf{c}$. Namely:

$$\mathbf{c}_0 = \mathbf{c} \bmod 2, \quad \text{and } \mathbf{c}_i = 2^{-i} \cdot \big((\mathbf{c} \bmod 2^{i+1}) - \sum_{j<i} 2^j \mathbf{c}_j\big) \text{ for } i > 0.$$

Then the bit-decomposition of $\mathbf{c}$ is the concatenation of all these vectors,

$$\hat{\mathbf{c}} = \mathsf{BitDecomp}(\mathbf{c}) \;\stackrel{\text{def}}{=}\; [\mathbf{c}_0 \mid \mathbf{c}_1 \mid \ \ldots \ \mid \mathbf{c}_l].$$

Clearly $\hat{\mathbf{c}}$ has low l_2 norm, since it is represented by a 0-1 vector, and we have $\langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle = \langle \mathbf{c}, \mathbf{s} \rangle$ over $\mathbb{Z}_q[X]$ (and therefore also over $\mathbb{Z}_q[X]/(X^m - 1)$). Switching keys from $\mathbf{s}$ to $\mathbf{t}$ is done simply by setting $\mathbf{c}' \leftarrow W\hat{\mathbf{c}} \bmod (X^m -$

$1, q)$. To see that this maintains our invariant, assume that for some $a \in \mathbb{Z}_p[X]/\Phi_m$ we have $G \cdot \langle \mathbf{c}, \mathbf{s} \rangle = p \cdot Gu + Ga \pmod{X^m - 1, q}$, where the coefficient vector of u has low l_2 norm. Then:

$$G \cdot \langle \mathbf{c}', \mathbf{t} \rangle = G \cdot \mathbf{t} W \hat{\mathbf{c}} \overset{(a)}{=} G \cdot \mathbf{t} \begin{bmatrix} \hat{s} + p\mathbf{v} \\ - \ 0 \ - \end{bmatrix} \hat{\mathbf{c}} \overset{(b)}{=} G \cdot \langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle + p \cdot G \cdot \langle \hat{\mathbf{c}}, \mathbf{v} \rangle$$
$$\overset{(c)}{=} G \cdot \langle \mathbf{c}, \mathbf{s} \rangle + p \cdot G \cdot \langle \hat{\mathbf{c}}, \mathbf{v} \rangle = p \cdot G \underbrace{(u + \langle \hat{\mathbf{c}}, \mathbf{v} \rangle)}_{u'} + Ga \pmod{X^m - 1, q},$$

where Equality (a) follows since $\mathbf{t}R = \mathbf{0} \pmod{\Phi_m, q}$ and therefore $G \cdot \mathbf{t}R = \mathbf{0} \pmod{X^m - 1, q}$, Equality (b) holds since the first entry of $\mathbf{t}$ is 1, and Equality (c) follows from $\langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle = \langle \mathbf{c}, \mathbf{s} \rangle$. Finally, since both $\mathbf{v}$ and $\hat{\mathbf{c}}$ have low l_2 norm, then over $\mathbb{Z}_q[X]/(X^m - 1)$ so has $\langle \hat{\mathbf{c}}, \mathbf{v} \rangle$ and therefore also $u' = \langle \hat{\mathbf{c}}, \mathbf{v} \rangle + u \bmod (X^m - 1, q)$.

E.6 Modulus switching

The modulus-switching procedure is exactly as in the BGV cryptosystem. Note that this procedure does not involve any mod-Φ_m or mod-$(X^m - 1)$ arithmetic: All we do is take a ciphertext vector $\mathbf{c}$ over $\mathbb{Z}_{q_i}[X]/(X^m - 1)$, scale it down by a factor q_{i+1}/q_i and round to get $\mathbf{c}' = \text{round}_{\mathbf{c}}(\frac{q_{i+1}}{q_i} \cdot \mathbf{c})$ such that $\mathbf{c}' \equiv \mathbf{c} \pmod{p}$. The reason that this works in our case is exactly as in BGV, our delayed reduction has no effect here.

E.7 Galois group actions

As described in Section 4.2, applying the action $X \to X^i$ on a ciphertext vector $\mathbf{c}$ over $\mathbb{Z}_q[X]/(X^m - 1)$ requires only a permutation of the coefficients in each of the elements of $\mathbf{c}$ (all which are degree-$(m-1)$ polynomials over $\mathbb{Z}_q$).

Assume that we have $G \cdot \langle \mathbf{c}, \mathbf{s} \rangle = p \cdot Gu + Ga \pmod{X^m - 1, q}$, and define $\mathbf{c}^{(i)}, u^{(i)}$ as what you get by applying the transformation $X \to X^i$ to $\mathbf{c}, u$, respectively, over $\mathbb{Z}_q[X]/(X^m - 1)$, and $\mathbf{s}^{(i)}, a^{(i)}$ as what you get by applying the transformation $X \to X^i$ to $\mathbf{s}, a$, respectively over $\mathbb{Z}_q[X]/\Phi_m$. Below we prove that if i, m are co-prime and also q, m are co-prime, then we have $G \cdot \langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \rangle = p \cdot Gu^{(i)} + Ga^{(i)} \pmod{X^m - 1, q}$.

Using $G = m \pmod{\Phi_m}$, and reducing modulo Φ_m the equality $G \cdot \langle \mathbf{c}, \mathbf{s} \rangle = p \cdot Gu + Ga$, we have $m \cdot \langle \mathbf{c}, \mathbf{s} \rangle = pm \cdot u + ma \pmod{\Phi_m, q}$. Since m, q are co-prime then multiplying by $m^{-1} \pmod{q}$ we get $\langle \mathbf{c}, \mathbf{s} \rangle = p \cdot u + a \pmod{\Phi_m, q}$. Namely, for some polynomial $k \in \mathbb{Z}_q[X]$ we have

$$\sum_j \mathbf{c}_j(X)\mathbf{s}_j(X) = p \cdot u(X) + a(X) + k(X)\Phi_m(X) \ \text{(equality in } \mathbb{Z}_q[X]), \tag{10}$$

and therefore also for every i

$$\sum_j \mathbf{c}_j(X^i)\mathbf{s}_j(X^i) = p \cdot u(X^i) + a(X^i) + k(X^i)\Phi_m(X^i) \ \text{(equality in } \mathbb{Z}_q[X]). \tag{11}$$

Equation (11) follows since the two sides of Equation (10) are identical as formal polynomials over $\mathbb{Z}_q$, and therefore they must coincide also as functions over any characteristic-q field. It follows that the functions on both sides of Equation (11) must also coincide over any characteristic-q field, and therefore the two sides must be identical as formal polynomials over $\mathbb{Z}_q$.

Recalling that if $i \in Z_m^*$ then $\Phi(X)$ divides $\Phi(X^i)$, we obtain

$$\left\langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \right\rangle = \sum_j \mathbf{c}_j(X^i)\mathbf{s}_j(X^i) = p \cdot u(X^i) + a(X^i) = p \cdot u^{(i)} + a^{(i)} \pmod{\Phi_m(X), q}.$$

Now we can multiply by G to "lift" the equality over to $\mathbb{Z}_q[X]/(X^m - 1)$ and we get

$$G \cdot \left\langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \right\rangle = p \cdot Gu^{(i)} + Ga^{(i)} \pmod{X^m - 1, q},$$

as needed. Observing that over $\mathbb{Z}_q[X]/(X^m - 1)$ the coefficient vectors of u and $u^{(i)}$ are just a permutation of each other (and hence have the same l_2 norm) we deduce that our invariant in maintained under the transformation $X \mapsto X^i$ whenever $i \in \mathbb{Z}_m^*$ and $m \in \mathbb{Z}_q^*$.

Homomorphic Evaluation of the AES Circuit
(Updated Implementation)

Craig Gentry
IBM Research

Shai Halevi
IBM Research

Nigel P. Smart
University of Bristol

January 29, 2015

Abstract

We describe a working implementation of leveled homomorphic encryption (with or without bootstrapping) that can evaluate the AES-128 circuit. This implementation is built on top of the HElib library, whose design was inspired by an early version of this work. Our main implementation (without bootstrapping) takes about 4 minutes and 3GB of RAM, running on a small laptop, to evaluate an entire AES-128 encryption operation. Using SIMD techniques, we can process upto 120 blocks in each such evaluation, yielding an amortized rate of just over 2 seconds per block.

For cases where further processing is needed after the AES computation, we describe a different setting that uses bootstrapping. We describe an implementation that lets us process 180 blocks in just over 18 minutes using 3.7GB of RAM on the same laptop, yielding amortized 6 seconds/block. We note that somewhat better amortized per-block cost can be obtained using "byte-slicing" (and maybe also "bit-slicing") implementations, at the cost of significantly slower wall-clock time for a single evaluation.

In this article we describe many of the optimizations that went into this implementation. These include both AES-specific optimizations, as well as several "generic" tools for FHE evaluation (which are incorporated in the HElib library). The generic tools include (among others) a different variant of the Brakerski-Vaikuntanathan key-switching technique that does not require reducing the norm of the ciphertext vector, and a method of implementing the Brakerski-Gentry-Vaikuntanathan modulus-switching transformation on ciphertexts in CRT representation.

Keywords. AES, Fully Homomorphic Encryption, Implementation

An early version of this work was published in CRYPTO 2012. The current report describes also more recent implementation work, done over the last two years.

For the early version, the first and second authors were partly sponsored by DARPA under agreement number FA8750-11-C-0096. The U.S. Government is authorized to reproduce and distribute reprints of the early version for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

For the same early version, the third author was sponsored by DARPA and AFRL under agreement number FA8750-11-2-0079. The same disclaimers as above apply. He is also supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant COED–EP/I03126X, and by a Royal Society Wolfson Merit Award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the European Commission or EPSRC.

Contents

1 Introduction

In his breakthrough result [13], Gentry demonstrated that fully-homomorphic encryption was theoretically possible, assuming the hardness of some problems in integer lattices. Since then, many different improvements have been made, for example authors have proposed new variants, improved efficiency, suggested other hardness assumptions, etc. Some of these works were accompanied by implementation [28, 14, 8, 29, 21, 9], but these implementations were either "proofs of concept" that can compute only one basic operation at a time (at great cost), or special-purpose implementations limited to evaluating very simple functions. In the early version of this work we reported on the first implementation powerful enough to support an "interesting real world circuit," specifically the AES-128 encryption operation. To this end, we implemented a variant of the leveled FHE-without-bootstrapping scheme of Brakerski, Gentry, and Vaikuntanathan [5] (BGV). In the current article we report on an updated implementation of the same circuit, using the "general purpose" open-source HElib library [18], whose design was inspired by that early version of our work. (As of December 2014, we made our new implementation available as part of HElib.)

Why AES? We chose to shoot for an evaluation of AES since it seems like a natural benchmark: AES is widely deployed and used extensively in security-aware applications (so it is "practically relevant" to implement it), and the AES circuit is nontrivial on one hand, but on the other hand not astronomical. Moreover the AES circuit has a regular (and quite "algebraic") structure , which is amenable to parallelism and optimizations. Indeed, for these same reasons AES is often used as a benchmark for implementations of protocols for secure multi-party computation (MPC), for example [26, 10, 19, 20]. Using the same yardstick to measure FHE and MPC protocols is quite natural, since these techniques target similar application domains and in some cases both techniques can be used to solve the same problem.

Beyond being a natural benchmark, homomorphic evaluation of AES decryption also has interesting applications: When data is encrypted under AES and we want to compute on that data, then homomorphic AES decryption would transform this AES-encrypted data into an FHE-encrypted data, and then we could perform whatever computation we wanted. (Such applications were alluded to in [21, 29, 6]).

Why BGV? Our implementation is based on the (ring-LWE-based) BGV cryptosystem [5], which is one of the few variants that seem the most likely to yield "somewhat practical" homomorphic encryption. Other variants are the NTRU-like cryptosystem of Lòpez-Alt et al. [23], the ring-LWE-based scale-invariant cryptosystem of Brakerski [4]. These three variants offer somewhat different implementation tradeoffs, but they all have similar performance characteristics. We don't expect the differences between these variants to be very significant, and moreover most of our optimizations for BGV are useful also for the other two variants. (Another interesting approach if to implement the newer cryptosystem of Gentry et al. [16], or some combination thereof.)

Contributions of this work. Our implementation is based on a variant of the BGV scheme [5, 7, 6] (based on ring-LWE [24]), using the techniques of Smart and Vercauteren (SV) [29] and Gentry, Halevi and Smart (GHS) [15], and we introduce many new optimizations. Some of our optimizations are specific to AES, these are described in Section 4. Most of our optimization, however, are more general-purpose and can be used for homomorphic evaluation of other circuits, these are described in Section 3.

Many of our general-purpose optimizations are aimed at reducing the number of FFTs and CRTs that we need to perform, by reducing the number of times that we need to convert polynomials between coefficient and evaluation representations. Since the cryptosystem is defined over a polynomial ring, many of

the operations involve various manipulation of integer polynomials, such as modular multiplications and additions and Frobenius maps. Most of these operations can be performed more efficiently in evaluation representation, when a polynomial is represented by the vector of values that it assumes in all the roots of the ring polynomial (for example polynomial multiplication is just point-wise multiplication of the evaluation values). On the other hand some operations in BGV-type cryptosystems (such as key switching and modulus switching) seem to require coefficient representation, where a polynomial is represented by listing all its coefficients.[1] Hence a "naive implementation" of FHE would need to convert the polynomials back and forth between the two representations, and these conversions turn out to be the most time-consuming part of the execution. In our implementation we keep ciphertexts in evaluation representation at all times, converting to coefficient representation only when needed for some operation, and then converting back.

We describe variants of key switching and modulus switching that can be implemented while keeping almost all the polynomials in evaluation representation. Our key-switching variant has another advantage, in that it significantly reduces the size of the key-switching matrices in the public key. This is particularly important since one limiting factor for evaluating "interesting" circuits is the ability to keep the key-switching matrices in memory. Other optimizations that we present are meant to reduce the number of modulus switching and key switching operations that we need to do.

Our Implementation and tests. Many of the optimizations described in this work were incorporated in the HElib C++ library, which is built on top of NTL (and GnuMP). We tested our implementation on a two years old Lenovo X230 laptop with Intel Core i5-3320M running at 2.6GHz, on which we run an Ubuntu 14.04 VM with 4GB of RAM and with the g++ compiler version 4.9.2. The detailed results of our tests are described in Section 4.4, the one-line summary is that we can evaluate AES-128 homomorphically on 120 blocks in 245 seconds on that commodity laptop. Also, if we need to incorporate extra processing then we can use bootstrapping and get evaluation on 180 blocks in under 18 minutes. All of our programs are single-threaded, so only one core was used in the computations.

We note that there are a multitude of optimizations that one can perform on our basic implementation. Most importantly, there are great gains to be had by making better use of parallelism: Unfortunately, the HElib library is not yet thread safe, which severely limits our ability to utilize the multi-core functionality of modern processors. Much of the work in homomorphic-AES is "embarrassingly parallelizable" and so we expect a fully parallel implementation to have a speedup factor roughly equal to the number of active cores (with parallelization opportunities not running our until perhaps 100x of current implementation). The byte-sliced and bit-sliced implementations (which we did not implement on top of HElib) obviously offer even more room for parallelism.

Organization. In Section 2 we review the main features of BGV-type cryptosystems [6, 5], and briefly survey the techniques for homomorphic computation on packed ciphertexts from SV and GHS [29, 15]. Then in Section 3 we describe our "general-purpose" optimizations on a high level, with additional details provided in Appendices A and B. A brief overview of AES and a high-level description and performance numbers is provided in Section 4.

[1] The need for coefficient representation ultimately stems from the fact that the noise in the ciphertexts is small in coefficient representation but not in evaluation representation.

2 Background

2.1 Notations and Mathematical Background

For an integer q we identify the ring $\mathbb{Z}/q\mathbb{Z}$ with the interval $(-q/2, q/2] \cap \mathbb{Z}$, and use $[z]_q$ to denote the reduction of the integer z modulo q into that interval. Our implementation utilizes polynomial rings defined by cyclotomic polynomials, $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$. The ring $\mathbb{A}$ is the ring of integers of a the mth cyclotomic number field $\mathbb{Q}(\zeta_m)$. We let $\mathbb{A}_q \stackrel{\text{def}}{=} \mathbb{A}/q\mathbb{A} = \mathbb{Z}[X]/(\Phi_m(X), q)$ for the (possibly composite) integer q, and we identify $\mathbb{A}_q$ with the set of integer polynomials of degree upto $\phi(m) - 1$ reduced modulo q.

Coefficient vs. Evaluation Representation. Let m, q be two integers such that $\mathbb{Z}/q\mathbb{Z}$ contains a primitive m-th root of unity, and denote one such primitive m-th root of unity by $\zeta \in \mathbb{Z}/q\mathbb{Z}$. Recall that the m'th cyclotomic polynomial splits into linear terms modulo q, $\Phi_m(X) = \prod_{i \in (\mathbb{Z}/m\mathbb{Z})^*} (X - \zeta^i) \pmod q$.

We consider two ways of representing an element $a \in \mathbb{A}_q$: Viewing a as a degree-$(\phi(m) - 1)$ polynomial, $a(X) = \sum_{i<\phi(m)} a_i X^i$, the *coefficient representation* of a just lists all the coefficients in order $\mathbf{a} = \langle a_0, a_1, \ldots, a_{\phi(m)-1} \rangle \in (\mathbb{Z}/q\mathbb{Z})^{\phi(m)}$. For the other representation we consider the values that the polynomial $a(X)$ assumes on all primitive m-th roots of unity modulo q, $b_i = a(\zeta^i) \bmod q$ for $i \in (\mathbb{Z}/m\mathbb{Z})^*$. The b_i's in order also yield a vector $\mathbf{b} \in (\mathbb{Z}/q\mathbb{Z})^{\phi(m)}$, which we call the *evaluation representation* of a. Clearly these two representations are related via $\mathbf{b} = V_m \cdot \mathbf{a}$, where V_m is the Vandermonde matrix over the primitive m-th roots of unity modulo q. We remark that for all i we have the equality $(a \bmod (X - \zeta^i)) = a(\zeta^i) = b_i$, hence the evaluation representation of a is just a polynomial Chinese-Remaindering representation.

In both representations, an element $a \in \mathbb{A}_q$ is represented by a $\phi(m)$-vector of integers in $\mathbb{Z}/q\mathbb{Z}$. If q is a composite then each of these integers can itself be represented either using the standard binary encoding of integers or using Chinese-Remaindering relative to the factors of q. We usually use the standard binary encoding for the coefficient representation and Chinese-Remaindering for the evaluation representation. (Hence the latter representation is really a *double CRT* representation, relative to both the polynomial factors of $\Phi_m(X)$ and the integer factors of q.)

2.2 BGV-type Cryptosystems

Our implementation uses a variant of the BGV cryptosystem due to Gentry, Halevi and Smart, specifically the one described in [15, Appendix D] (in the full version). In this cryptosystem both ciphertexts and secret keys are vectors over the polynomial ring $\mathbb{A}$, and the native plaintext space is the space of binary polynomials $\mathbb{A}_2$. (More generally it could be $\mathbb{A}_p$ for some fixed $p \geq 2$, but in our case we will always use $\mathbb{A}_2$.)

At any point during the homomorphic evaluation there is some "current integer modulus q" and "current secret key $\mathbf{s}$", that change from time to time. A ciphertext $\mathbf{c}$ is decrypted using the current secret key $\mathbf{s}$ by taking inner product over $\mathbb{A}_q$ (with q the current modulus) and then reducing the result modulo 2 *in coefficient representation*. Namely, the decryption formula is

$$a \leftarrow [\ \underbrace{[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q}_{\text{noise}}\ ]_2 \,. \tag{1}$$

The polynomial $[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q$ is called the "noise" in the ciphertext $\mathbf{c}$. Informally, $\mathbf{c}$ is a *valid ciphertext* with respect to secret key $\mathbf{s}$ and modulus q if this noise has "sufficiently small norm" relative to q. The meaning of "sufficiently small norm" is whatever is needed to ensure that the noise does not wrap around q when performing homomorphic operations, in our implementation we keep the norm of the noise always below some pre-set bound (which is determined in Appendix C.2).

Following [24, 15], the specific norm that we use to evaluate the magnitude of the noise is the "canonical embedding norm reduced mod q", specifically we use the conventions as described in [15, Appendix D] (in the full version). This is useful to get smaller parameters, but for the purpose of presentation the reader can think of the norm as the Euclidean norm of the noise in coefficient representation. More details are given in the Appendices. We refer to the norm of the noise as *the noise magnitude*.

The central feature of BGV-type cryptosystems is that the current secret key and modulus evolve as we apply operations to ciphertexts. We apply five different operations to ciphertexts during homomorphic evaluation. Three of them — addition, multiplication, and automorphism — are "semantic operations" that we use to evolve the plaintext data which is encrypted under those ciphertexts. The other two operations — key-switching and modulus-switching — are used for "maintenance": These operations do not change the plaintext at all, they only change the current key or modulus (respectively), and they are mainly used to control the complexity of the evaluation. Below we briefly describe each of these five operations on a high level. For the sake of self-containment, we also describe key generation and encryption in Appendix B. More detailed description can be found in [15, Appendix D].

Addition. Homomorphic addition of two ciphertext vectors with respect to the same secret key and modulus q is done just by adding the vectors over $\mathbb{A}_q$. If the two arguments were encrypting the plaintext polynomials $a_1, a_2 \in \mathbb{A}_2$ then the sum will be an encryption of $a_1 + a_2 \in \mathbb{A}_2$. This operation has no effect on the current modulus or key, and the norm of the noise is at most the sum of norms from the noise in the two arguments.

Multiplication. Homomorphic multiplication is done via tensor product over $\mathbb{A}_q$. In principle, if the two arguments have dimension n over $\mathbb{A}_q$ then the product ciphertext has dimension n^2, each entry in the output computed as the product of one entry from the first argument and one entry from the second.[2]

This operation does not change the current modulus, but it changes the current key: If the two input ciphertexts are valid with respect to the dimension-n secret key vector $\mathbf{s}$, encrypting the plaintext polynomials $a_1, a_2 \in \mathbb{A}_2$, then the output is valid with respect to the dimension-n^2 secret key $\mathbf{s}'$ which is the tensor product of $\mathbf{s}$ with itself, and it encrypts the polynomial $a_1 \cdot a_2 \in \mathbb{A}_2$. The norm of the noise in the product ciphertext can be bounded in terms of the product of norms of the noise in the two arguments. For our choice of norm function, the norm of the product is no larger than the product of the norms of the two arguments.

Key Switching. The public key of BGV-type cryptosystems includes additional components to enable converting a valid ciphertext with respect to one key into a valid ciphertext encrypting the same plaintext with respect to another key. For example, this is used to convert the product ciphertext which is valid with respect to a high-dimension key back to a ciphertext with respect to the original low-dimension key.

To allow conversion from dimension-n' key $\mathbf{s}'$ to dimension-n key $\mathbf{s}$ (both with respect to the same modulus q), we include in the public key a matrix $W = W[\mathbf{s}' \to \mathbf{s}]$ over $\mathbb{A}_q$, where the i'th column of W is roughly an encryption of the i'th entry of $\mathbf{s}'$ with respect to $\mathbf{s}$ (and the current modulus). Then given a valid ciphertext $\mathbf{c}'$ with respect to $\mathbf{s}'$, we roughly compute $\mathbf{c} = W \cdot \mathbf{c}'$ to get a valid ciphertext with respect to $\mathbf{s}$.

In some more detail, the BGV key switching transformation first ensures that the norm of the ciphertext $\mathbf{c}'$ itself is sufficiently low with respect to q. In [5] this was done by working with the binary encoding of $\mathbf{c}'$, and one of our main optimization in this work is a different method for achieving the same goal (cf. Section 3.1). Then, if the i'th entry in $\mathbf{s}'$ is $\mathfrak{s}'_i \in \mathbb{A}$ (with norm smaller than q), then the i'th column of $W[\mathbf{s}' \to \mathbf{s}]$ is an n-vector $\mathbf{w}_i$ such that $[\langle \mathbf{w}_i, \mathbf{s} \rangle \bmod \Phi_m(X)]_q = 2e_i + \mathfrak{s}'_i$ for a low-norm polynomial

[2] It was shown in [7] that over polynomial rings this operation can be implemented while increasing the dimension only to $2n-1$ rather than to n^2.

$e_i \in \mathbb{A}$. Denoting $\mathbf{e} = (e_1, \ldots, e_{n'})$, this means that we have $\mathbf{s}W = \mathbf{s}' + 2\mathbf{e}$ over $\mathbb{A}_q$. For any ciphertext vector $\mathbf{c}'$, setting $\mathbf{c} = W \cdot \mathbf{c}' \in \mathbb{A}_q$ we get the equation

$$[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q = [\mathbf{s}W\mathbf{c}' \bmod \Phi_m(X)]_q = [\langle \mathbf{c}', \mathbf{s}' \rangle + 2\langle \mathbf{c}', \mathbf{e} \rangle \bmod \Phi_m(X)]_q$$

Since $\mathbf{c}'$, $\mathbf{e}$, and $[\langle \mathbf{c}', \mathbf{s}' \rangle \bmod \Phi_m(X)]_q$ all have low norm relative to q, then the addition on the right-hand side does not cause a wrap around q, hence we get $[[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q]_2 = [[\langle \mathbf{c}', \mathbf{s}' \rangle \bmod \Phi_m(X)]_q]_2$, as needed. The key-switching operation changes the current secret key from $\mathbf{s}'$ to $\mathbf{s}$, and does not change the current modulus. The norm of the noise is increased by at most an additive factor of $2\| \langle \mathbf{c}', \mathbf{e} \rangle \|$.

Modulus Switching. The modulus switching operation is intended to reduce the norm of the noise, to compensate for the noise increase that results from all the other operations. To convert a ciphertext $\mathbf{c}$ with respect to secret key $\mathbf{s}$ and modulus q into a ciphertext $\mathbf{c}'$ encrypting the same thing with respect to the same secret key but modulus q', we roughly just scale $\mathbf{c}$ by a factor q'/q (thus getting a fractional ciphertext), then round appropriately to get back an integer ciphertext. Specifically $\mathbf{c}'$ is a ciphertext vector satisfying (a) $\mathbf{c}' = \mathbf{c} \pmod 2$, and (b) the "rounding error term" $\tau \stackrel{\text{def}}{=} \mathbf{c}' - (q'/q)\mathbf{c}$ has low norm. Converting $\mathbf{c}$ to $\mathbf{c}'$ is easy in coefficient representation, and one of our optimizations is a method for doing the same in evaluation representation (cf. Section 3.2) This operation leaves the current key $\mathbf{s}$ unchanged, changes the current modulus from q to q', and the norm of the noise is changed as $\|n'\| \leq (q'/q)\|n\| + \|\tau \cdot s\|$. Note that if the key $\mathbf{s}$ has low norm and q' is sufficiently smaller than q, then the noise magnitude decreases by this operation.

A BGV-type cryptosystem has a chain of moduli, $q_0 < q_1 \cdots < q_{L-1}$, where fresh ciphertexts are with respect to the largest modulus q_{L-1}. During homomorphic evaluation every time the (estimated) noise grows too large we apply modulus switching from q_i to q_{i-1} in order to decrease it back. Eventually we get ciphertexts with respect to the smallest modulus q_0, and we cannot compute on them anymore (except by using bootstrapping).

Automorphisms. In addition to adding and multiplying polynomials, another useful operation is converting the polynomial $a(X) \in \mathbb{A}$ to $a^{(i)}(X) \stackrel{\text{def}}{=} a(X^i) \bmod \Phi_m(X)$. Denoting by κ_i the transformation $\kappa_i : a \mapsto a^{(i)}$, it is a standard fact that the set of transformations $\{\kappa_i : i \in (\mathbb{Z}/m\mathbb{Z})^*\}$ forms a group under composition (which is the Galois group $\mathcal{G}\mathsf{al}(\mathbb{Q}(\zeta_m)/\mathbb{Q})$), and this group is isomorphic to $(\mathbb{Z}/m\mathbb{Z})^*$. In [5, 15] it was shown that applying the transformations κ_i to the plaintext polynomials is very useful, some more examples of its use can be found in our Section 4.

Denoting by $\mathbf{c}^{(i)}$, $\mathbf{s}^{(i)}$ the vector obtained by applying κ_i to each entry in $\mathbf{c}, \mathbf{s}$, respectively, it was shown in [5, 15] that if $\mathbf{s}$ is a valid ciphertext encrypting a with respect to key $\mathbf{s}$ and modulus q, then $\mathbf{c}^{(i)}$ is a valid ciphertext encrypting $a^{(i)}$ with respect to key $\mathbf{s}^{(i)}$ and the same modulus q. Moreover the norm of noise remains the same under this operation. We remark that we can apply key-switching to $\mathbf{c}^{(i)}$ in order to get an encryption of $a^{(i)}$ with respect to the original key $\mathbf{s}$.

2.3 Computing on Packed Ciphertexts

Smart and Vercauteren observed [28, 29] that the plaintext space $\mathbb{A}_2$ can be viewed as a vector of "plaintext slots", by an application the polynomial Chinese Remainder Theorem. Specifically, if the ring polynomial $\Phi_m(X)$ factors modulo 2 into a product of irreducible factors $\Phi_m(X) = \prod_{j=0}^{\ell-1} F_j(X) \pmod 2$, then a plaintext polynomial $a(X) \in \mathbb{A}_2$ can be viewed as encoding ℓ different small polynomials, $a_j = a \bmod F_j$. Just like for integer Chinese Remaindering, addition and multiplication in $\mathbb{A}_2$ correspond to element-wise addition and multiplication of the vectors of slots.

The effect of the automorphisms is a little more involved. When i is a power of two then the transformations $\kappa_i : a \mapsto a^{(i)}$ is just applied to each slot separately. When i is not a power of two the transformation κ_i has the effect of roughly shifting the values between the different slots. For example, for some parameters we could get a cyclic shift of the vector of slots: If a encodes the vector $(a_0, a_1, \ldots, a_{\ell-1})$, then $\kappa_i(a)$ (for some i) could encode the vector $(a_{\ell-1}, a_0, \ldots, a_{\ell-2})$. This was used in [15] to devise efficient procedures for applying arbitrary permutations to the plaintext slots.

We note that the values in the plaintext slots are not just bits, rather they are polynomials modulo the irreducible F_j's, so they can be used to represents elements in extension fields GF(2^d). In particular, in our AES implementations we used the plaintext slots to hold elements of GF(2^8), and encrypt one byte of the AES state in each slot. Then we can use an adaption of the techniques from [15] to permute the slots when performing the AES row-shift and column-mix.

3 General-Purpose Optimizations

Below we summarize our optimizations that are not tied directly to the AES circuit and can be used also in homomorphic evaluation of other circuits. Underlying many of these optimizations is our choice of keeping ciphertext and key-switching matrices in evaluation (double-CRT) representation. Roughly speaking, our chain of moduli is defined via a set of same-size primes, $p_0, p_1, p_2, \ldots$, chosen such that $\mathbb{Z}/p_i\mathbb{Z}$ has m'th roots of unity. (In other words, $m|p_i - 1$ for all i.) For $i = 0, \ldots, L-1$ we then define our i'th modulus as $q_i = \prod_{j=0}^{i} p_i$. To gain efficiency, we actually choose p_0 to be half the bit-size of the other p_i's, and so the odd indexed moduli in the chain are a product of the primes starting at p_0 ($q_i = \prod_{i=0}^{\lfloor i/2 \rfloor} p_i$) and the even-indexed moduli are products that do not include p_0 ($q_i = \prod_{i=1}^{i/2} p_i$). In our implementation the half-sized prime has 23-25 bits (and the full-sized primes therefore have 46-50 bits). For easy of exposition, however, in the rest of this report we ignore this "half-sized" prime and describe all our optimizations as if we were using only a chain of same-size primes.

In the t-th level of the scheme we have ciphertexts consisting of elements in $\mathbb{A}_{q_t}$ (i.e., polynomials modulo $(\Phi_m(X), q_t)$). We represent an element $c \in \mathbb{A}_{q_t}$ by a $\phi(m) \times (t+1)$ "matrix" of its evaluations at the primitive m-th roots of unity modulo the primes $p_0, \ldots, p_t$. Computing this representation from the coefficient representation of c involves reducing c modulo the p_i's and then $t+1$ invocations of the FFT algorithm, modulo each of the p_i (picking only the FFT coefficients corresponding to $(\mathbb{Z}/m\mathbb{Z})^*$). To convert back to coefficient representation we invoke the inverse FFT algorithm, each time padding the $\phi(m)$-vector of evaluation point with $m - \phi(m)$ zeros (for the evaluations at the non-primitive roots of unity). This yields the coefficients of the polynomials modulo $(X^m - 1, p_i)$ for $i = 0, \ldots, t$, we then reduce each of these polynomials modulo $(\Phi_m(X), p_i)$ and apply Chinese Remainder interpolation. We stress that we try to perform these transformations as rarely as we can.

3.1 A New Variant of Key Switching

As described in Section 2, the key-switching transformation introduces an additive factor of $2\langle \mathbf{c}', \mathbf{e} \rangle$ in the noise, where $\mathbf{c}'$ is the input ciphertext and $\mathbf{e}$ is the noise component in the key-switching matrix. To keep the noise magnitude below the modulus q, it seems that we need to ensure that the ciphertext $\mathbf{c}'$ itself has low norm. In BGV [5] this was done by representing $\mathbf{c}'$ as a fixed linear combination of small vectors, i.e. $\mathbf{c}' = \sum_i 2^i \mathbf{c}'_i$ with $\mathbf{c}'_i$ the vector of i'th bits in $\mathbf{c}'$. Considering the high-dimension ciphertext $\mathbf{c}^* = (\mathbf{c}'_0|\mathbf{c}'_1|\mathbf{c}'_2|\cdots)$ and secret key $\mathbf{s}^* = (\mathbf{s}'|2\mathbf{s}'|4\mathbf{s}'|\cdots)$, we note that we have $\langle \mathbf{c}^*, \mathbf{s}^* \rangle = \langle \mathbf{c}', \mathbf{s}' \rangle$, and $\mathbf{c}^*$ has low norm (since it consists of 0-1 polynomials). BGV therefore included in the public key the matrix

$W = W[\mathbf{s}^* \to \mathbf{s}]$ (rather than $W[\mathbf{s}' \to \mathbf{s}]$), and had the key-switching transformation computes $\mathbf{c}^*$ from $\mathbf{c}'$ and sets $\mathbf{c} = W \cdot \mathbf{c}^*$.

When implementing key-switching, there are two drawbacks to the above approach. First, this increases the dimension (and hence the size) of the key switching matrix. This drawback is fatal when evaluating deep circuits, since having enough memory to keep the key-switching matrices turns out to be a limiting factor in our ability to evaluate such circuits. In addition, for this key-switching we must first convert $\mathbf{c}'$ to coefficient representation (in order to compute the $\mathbf{c}'_i$'s), then convert each of the $\mathbf{c}'_i$'s back to evaluation representation before multiplying by the key-switching matrix. In level t of the circuit, this seem to require $\Omega(t \log q_t)$ FFTs.

In this work we propose a different variant: Rather than manipulating $\mathbf{c}'$ to decrease its norm, we instead temporarily increase the modulus q. We recall that for a valid ciphertext $\mathbf{c}'$, encrypting plaintext a with respect to $\mathbf{s}'$ and q, we have the equality $\langle \mathbf{c}', \mathbf{s}' \rangle = 2e' + a$ over A_q, for a low-norm polynomial e'. This equality, we note, implies that for every odd integer p we have the equality $\langle \mathbf{c}', p\mathbf{s}' \rangle = 2e'' + a$, *holding over* A_{pq}, for the "low-norm" polynomial e'' (namely $e'' = p \cdot e' + \frac{p-1}{2}a$). Clearly, when considered relative to secret key $p\mathbf{s}$ and modulus pq, the noise in $\mathbf{c}'$ is p times larger than it was relative to $\mathbf{s}$ and q. However, since the modulus is also p times larger, we maintain that the noise has norm sufficiently smaller than the modulus. In other words, $\mathbf{c}'$ is still a valid ciphertext that encrypts the same plaintext a with respect to secret key $p\mathbf{s}$ and modulus pq. By taking p large enough, we can ensure that the norm of $\mathbf{c}'$ (which is independent of p) is sufficiently small relative to the modulus pq.

We therefore include in the public key a matrix $W = W[p\mathbf{s}' \to \mathbf{s}]$ modulo pq for a large enough odd integer p. (Specifically we need $p \approx q\sqrt{m}$.) Given a ciphertext $\mathbf{c}'$, valid with respect to $\mathbf{s}$ and q, we apply the key-switching transformation simply by setting $\mathbf{c} = W \cdot \mathbf{c}'$ over $\mathbb{A}_{pq}$. The additive noise term $\langle \mathbf{c}', \mathbf{e} \rangle$ that we get is now small enough relative to our large modulus pq, thus the resulting ciphertext $\mathbf{c}$ is valid with respect to $\mathbf{s}$ and pq. We can now switch the modulus back to q (using our modulus switching routine), hence getting a valid ciphertext with respect to $\mathbf{s}$ and q.

We note that even though we no longer break $\mathbf{c}'$ into its binary encoding, it seems that we still need to recover it in coefficient representation in order to compute the evaluations of $\mathbf{c}' \bmod p$. However, since we do not increase the dimension of the ciphertext vector, this procedure requires only $O(t)$ FFTs in level t (vs. $O(t \log q_t) = O(t^2)$ for the original BGV variant). Also, the size of the key-switching matrix is reduced by roughly the same factor of $\log q_t$.

Our new variant comes with a price tag, however: We use key-switching matrices relative to a larger modulus, but still need the noise term in this matrix to be small. This means that the LWE problem underlying this key-switching matrix has larger ratio of modulus/noise, implying that we need a larger dimension to get the same level of security than with the original BGV variant. In fact, since our modulus is more than squared (from q to pq with $p > q$), the dimension is increased by more than a factor of two. This translates to more than doubling of the key-switching matrix, partly negating the size and running time advantage that we get from this variant.

Of course, one can also use a hybrid of the two approaches: we can decrease the norm of $\mathbf{c}'$ only somewhat by breaking it into a few digits (as opposed to binary bits as in [5]), and then increase the modulus somewhat until it is large enough relative to the smaller norm of $\mathbf{c}'$. The HElib implementation indeed let us break $\mathbf{c}$ to any number of digits, upto the number of primes in the chain, and in our experiments we used anywhere between 3 and 6 digits to get the right level of security for the different settings.

3.2 Modulus Switching in Evaluation Representation

Given an element $c \in \mathbb{A}_{q_t}$ in evaluation (double-CRT) representation relative to $q_t = \prod_{j=0}^{t} p_j$, we want to modulus-switch to q_{t-1} – i.e., scale down by a factor of p_t; we call this operation $\mathsf{Scale}(c, q_t, q_{t-1})$. The output should be $c' \in \mathbb{A}$, represented via the same double-CRT format (with respect to $p_0, \ldots, p_{t-1}$), such that (a) $c' \equiv c \pmod 2$, and (b) the "rounding error term" $\tau = c' - (c/p_t)$ has a very low norm. As p_t is odd, we can equivalently require that the element $c^\dagger \stackrel{\text{def}}{=} p_t \cdot c'$ satisfy

(i) $c^\dagger$ is divisible by p_t,

(ii) $c^\dagger \equiv c \pmod 2$, and

(iii) $c^\dagger - c$ (which is equal to $p_t \cdot \tau$) has low norm.

Rather than computing c' directly, we will first compute $c^\dagger$ and then set $c' \leftarrow c^\dagger / p_t$. Observe that once we compute $c^\dagger$ in double-CRT format, it is easy to output also c' in double-CRT format: given the evaluations for $c^\dagger$ modulo p_j ($j < t$), simply multiply them by $p_t^{-1} \bmod p_j$. The algorithm to output $c^\dagger$ in double-CRT format is as follows:

1. Set $\bar{c}$ to be the coefficient representation of $c \bmod p_t$. (Computing this requires a single "small FFT" modulo the prime p_t.)

2. Add or subtract p_t from every odd coefficient of $\bar{c}$, thus obtaining a polynomial δ with coefficients in $(-p_t, p_t]$ such that $\delta \equiv \bar{c} \equiv c \pmod{p_t}$ and $\delta \equiv 0 \pmod 2$.

3. Set $c^\dagger = c - \delta$, and output it in double-CRT representation.

   Since we already have c in double-CRT representation, we only need the double-CRT representation of δ, which requires t more "small FFTs" modulo the p_j's.

As all the coefficients of $c^\dagger$ are within p_t of those of c, the "rounding error term" $\tau = (c^\dagger - c)/p_t$ has coefficients of magnitude at most one, hence it has low norm.

The procedure above uses $t + 1$ small FFTs in total. This should be compared to the naive method of just converting everything to coefficient representation modulo the primes ($t + 1$ FFTs), CRT-interpolating the coefficients, dividing and rounding appropriately the large integers (of size $\approx q_t$), CRT-decomposing the coefficients, and then converting back to evaluation representation ($t + 1$ more FFTs). The above approach makes explicit use of the fact that we are working in a plaintext space modulo 2; in Appendix D we present a technique which works when the plaintext space is defined modulo a larger modulus.

3.3 Dynamic Noise Management

As described in the literature, BGV-type cryptosystems tacitly assume that each homomorphic operation operation is followed a modulus switch to reduce the noise magnitude. In our implementation, however, we attach to each ciphertext an estimate of the noise magnitude in that ciphertext, and use these estimates to decide dynamically when a modulus switch must be performed.

Each modulus switch consumes a level, and hence a goal is to reduce, over a computation, the number of levels consumed. By paying particular attention to the parameters of the scheme, and by carefully analyzing how various operations affect the noise, we are able to control the noise much more carefully than in prior work. In particular, we note that modulus-switching is really only necessary just prior to multiplication (when the noise magnitude is about to get squared), in other times it is acceptable to keep the ciphertexts at a higher level (with higher noise).

4 Homomorphic Evaluation of AES

Next we describe our homomorphic implementation of AES-128. Our main impelemntation is "packed", namely the entire AES state is packed in just one ciphertext. Two other possible implementations (of byte-slice and bit-slice AES) are described later in Section 4.2. We note that in our earlier work we implemented all htree versions, but in the newer work we only re-implemented the "packed" version.

A Brief Overview of AES. The AES-128 cipher consists of ten applications of the same keyed round function (with different round keys). The round function operates on a 4×4 matrix of bytes, which are sometimes considered as element of $\mathbb{F}_{2^8}$. The basic operations that are performed during the round function are AddKey, SubBytes, ShiftRows, MixColumns. The AddKey is simply an XOR operation of the current state with 16 bytes of key; the SubBytes operation consists of an inversion in the field $\mathbb{F}_{2^8}$ followed by a fixed $\mathbb{F}_2$-affine map on the bits of the element; the ShiftRows rotates the entries in the row i of the 4×4 matrix by $i-1$ places to the left; finally the MixColumns operations pre-multiplies the state matrix by a fixed 4×4 matrix.

Our Packed Representation of the AES state. For our implementation we chose the native plaintext space of our homomorphic encryption so as to support operations on the finite field $\mathbb{F}_{2^8}$. To this end we choose our ring polynomial as $\Phi_m(X)$ that factors modulo 2 into degree-d irreducible polynomials such that $8|d$. (In other words, the smallest integer d such that $m|(2^d-1)$ is divisible by 8.) This means that our plaintext slots can hold elements of $\mathbb{F}_{2^d}$, and in particular we can use them to hold elements of $\mathbb{F}_{2^8}$ which is a sub-field of $\mathbb{F}_{2^d}$. Since we have $\ell = \phi(m)/d$ plaintext slots in each ciphertext, we can represent upto $\lfloor \ell/16 \rfloor$ complete AES state matrices per ciphertext.

Moreover, we choose our parameter m so that there exists an element $g \in \mathbb{Z}_m^*$ that has order 16 in both $\mathbb{Z}_m^*$ and the quotient group $\mathbb{Z}_m^*/\langle 2 \rangle$. This condition means that if we put 16 plaintext bytes in slots $t, tg, tg^2, tg^3, \ldots$ (for some $t \in \mathbb{Z}_m^*$), then the conjugation operation $X \mapsto X^g$ implements a cyclic right shift over these sixteen plaintext bytes. Below we denote the vector of plaintext slots by $a = (\alpha_i)_{i=1}^{\ell}$, with each $\alpha_i \in \mathbb{F}_{2^8}$. We place the 16 bytes of the AES state in plaintext slots using column-first ordering, namely we have

$$a \approx [\alpha_{00}\alpha_{10}\alpha_{20}\alpha_{30}\alpha_{01}\alpha_{11}\alpha_{21}\alpha_{31}\alpha_{02}\alpha_{12}\alpha_{22}\alpha_{32}\alpha_{03}\alpha_{13}\alpha_{23}\alpha_{33}],$$

representing the input plaintext matrix

$$A = \big(\alpha_{ij}\big)_{i,j} \;=\; \begin{pmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix}.$$

4.1 Homomorphic Evaluation of the Basic Operations

We now examine each AES operation in turn, and describe how it is implemented homomorphically.

4.1.1 AddKey and SubBytes

The AddKey is just a simple addition of ciphertexts, which yields a 4×4 matrix of bytes in the input to the SubBytes operation.

During S-box lookup, each plaintext byte α_{ij} should be replaced by $\beta_{ij} = S(\alpha_{ij})$, where $S(\cdot)$ is a fixed permutation on the bytes. Specifically, $S(x)$ is obtained by first computing $y = x^{-1}$ in $\mathbb{F}_{2^8}$ (with 0 mapped to 0), then applying a bitwise affine transformation $z = T(y)$ where elements in $\mathbb{F}_{2^8}$ are treated as bit strings with representation polynomial $G(X) = x^8 + x^4 + x^3 + x + 1$.

We implement $\mathbb{F}_{2^8}$ inversion followed by the $\mathbb{F}_2$ affine transformation using the Frobenius automorphisms, $X \longrightarrow X^{2^j}$. Recall that the transformation $\kappa_{2^j}(a(X)) = (a(X^{2^j}) \bmod \Phi_m(X))$ is applied separately to each slot, hence we can use it to transform the vector $(\alpha_i)_{i=1}^{\ell}$ into $(\alpha_i^{2^j})_{i=1}^{\ell}$. We note that applying the Frobenius automorphisms to ciphertexts has almost no influence on the noise magnitude, and hence it does not consume any levels.[3]

Inversion over $\mathbb{F}_{2^8}$ is done using essentially the same procedure as Algorithm 2 from [27] for computing $\beta = \alpha^{-1} = \alpha^{254}$. This procedure takes only three Frobenius automorphisms and four multiplications, arranged in a depth-3 circuit (see details below.) To apply the AES $\mathbb{F}_2$ affine transformation, we use the fact that any $\mathbb{F}_2$ affine transformation can be computed as a $\mathbb{F}_{2^8}$ affine transformation over the conjugates. Thus there are constants $\gamma_0, \gamma_1, \ldots, \gamma_7, \delta \in \mathbb{F}_{2^8}$ such that the AES affine transformation $T_{\mathrm{AES}}(\cdot)$ can be expressed as $T_{\mathrm{AES}}(\beta) = \delta + \sum_{j=0}^{7} \gamma_j \cdot \beta^{2^j}$ over $\mathbb{F}_{2^8}$. We therefore again apply the Frobenius automorphisms to compute eight ciphertexts encrypting the polynomials $\kappa_{2^j}(b)$ for $j = 0, 1, \ldots, 7$, and take the appropriate linear combination (with coefficients the γ_j's) to get an encryption of the vector $(T_{\mathrm{AES}}(\alpha_i^{-1}))_{i=1}^{\ell}$. For our parameters, a multiplication-by-constant operation consumes roughly half a level in terms of added noise.

One subtle implementation detail to note here, is that although our plaintext slots all hold elements of the same field $\mathbb{F}_{2^8}$, they hold these elements with respect to different polynomial encodings. The AES affine transformation, on the other hand, is defined with respect to one particular fixed polynomial encoding. This means that we must implement in the i'th slot not the affine transformation $T_{\mathrm{AES}}(\cdot)$ itself but rather the projection of this transformation onto the appropriate polynomial encoding: When we take the affine transformation of the eight ciphertexts encrypting $b_j = \kappa_{2^j}(b)$, we therefore multiply the encryption of b_j not by a constant that has γ_j in all the slots, but rather by a constant that has in slot i the projection of γ_j to the polynomial encoding of slot i.

Below we provide a pseudo-code description of our S-box lookup implementation, together with an approximation of the levels that are consumed by these operations.

	Level	
Input: ciphertext $\mathfrak{c}$	t	
// Compute $\mathfrak{c}_{254} = \mathfrak{c}^{-1}$		
1. $\mathfrak{c}_2 \leftarrow \mathfrak{c} \gg 2$	t	// Frobenius $X \mapsto X^2$
2. $\mathfrak{c}_3 \leftarrow \mathfrak{c} \times \mathfrak{c}_2$	$t-1$	// Multiplication
3. $\mathfrak{c}_{12} \leftarrow \mathfrak{c}_3 \gg 4$	$t-1$	// Frobenius $X \mapsto X^4$
4. $\mathfrak{c}_{14} \leftarrow \mathfrak{c}_{12} \times \mathfrak{c}_2$	$t-2$	// Multiplication
5. $\mathfrak{c}_{15} \leftarrow \mathfrak{c}_{12} \times \mathfrak{c}_3$	$t-2$	// Multiplication
6. $\mathfrak{c}_{240} \leftarrow \mathfrak{c}_{15} \gg 16$	$t-2$	// Frobenius $X \mapsto X^{16}$
7. $\mathfrak{c}_{254} \leftarrow \mathfrak{c}_{240} \times \mathfrak{c}_{14}$	$t-3$	// Multiplication
// Affine transformation over $\mathbb{F}_2$		
8. $\mathfrak{c}'_{2^j} \leftarrow \mathfrak{c}_{254} \gg 2^j$ for $j = 0, 1, 2, \ldots, 7$	$t-3$	// Frobenius $X \mapsto X^{2^j}$
9. $\mathfrak{c}'' \leftarrow \gamma + \sum_{j=0}^{7} \gamma_j \times \mathfrak{c}'_{2^j}$	$t-3.5$	// Linear combination over $\mathbb{F}_{2^8}$

[3] It does increase the noise magnitude somewhat, because we need to do key switching after these automorphisms. But this is only a small influence, and we will ignore it here.

4.1.2 ShiftRows **and** MixColumns

As commonly done, we lump together the ShiftRows/MixColumns operations, viewing both as a single linear transformation over vectors from $(\mathbb{F}_{2^8})^{16}$. As mentioned above, by a careful choice of the parameter m and the placement of the AES state bytes in our plaintext slots, we can implement a rotation-by-i of the rows of the AES matrix as a single automorphism operations $X \mapsto X^{g^i}$ (for some element $g \in (\mathbb{Z}/m\mathbb{Z})^*$). Given the ciphertext $\mathfrak{c}''$ after the SubBytes step, we use these operations in conjunction with ℓ-SELECT operations (as described in [15]) to compute four ciphertexts corresponding to the appropriate permutations of the 16 bytes (in each of the $\ell/16$ different input blocks). These four ciphertexts are combined via a linear operation (with coefficients $1, X$, and $(1+X)$) to obtain the final result of this round function.

Moreover, the multiply-by-constant operations implied by ℓ-SELECT can be folded into the multiply-by-constant operations of the linear transformations, hence the entire shift-row/mix-column operation consumes only 1/2 level in terms of noise. Finally, it is possible to implement the entire procedure using only six rotation operations, as described next. Recall our column-byte-ordering of the AES state:

$$a \approx [\alpha_{00}\alpha_{10}\alpha_{20}\alpha_{30}\alpha_{01}\alpha_{11}\alpha_{21}\alpha_{31}\alpha_{02}\alpha_{12}\alpha_{22}\alpha_{32}\alpha_{03}\alpha_{13}\alpha_{23}\alpha_{33}]$$
$$A = \begin{pmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix}.$$

We apply to the state vector a three right-rotations by 11, 6, and 1 positions to get the three vectors a_{11}, a_6, a_1 representing the matrices A_{11}, A_6, A_1, respectively:

$$a_{11} \approx [\alpha_{11}\alpha_{21}\alpha_{31}\ldots\alpha_{30}\alpha_{01}] \qquad a_6 \approx [\alpha_{22}\alpha_{32}\alpha_{03}\ldots\alpha_{02}\alpha_{12}] \qquad a_1 \approx [\alpha_{33}\alpha_{00}\alpha_{10}\ldots\alpha_{13}\alpha_{23}]$$
$$A_{11} = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{10} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{20} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{30} \\ \alpha_{02} & \alpha_{03} & \alpha_{00} & \alpha_{01} \end{pmatrix} \quad A_6 = \begin{pmatrix} \alpha_{22} & \alpha_{23} & \alpha_{20} & \alpha_{21} \\ \alpha_{32} & \alpha_{33} & \alpha_{30} & \alpha_{31} \\ \alpha_{03} & \alpha_{00} & \alpha_{01} & \alpha_{02} \\ \alpha_{13} & \alpha_{10} & \alpha_{11} & \alpha_{12} \end{pmatrix} \quad A_1 = \begin{pmatrix} \alpha_{33} & \alpha_{30} & \alpha_{31} & \alpha_{32} \\ \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \end{pmatrix}$$

Considering the top row in the four matrices (consisting of the bytes in positions 0,4,8,12), we see that we get exactly the four rows of the matrix after the shift-row operations. Hence these four bytes in the four matrices are exactly aligned so we can use SIMD operations to compute the column-mix operations. We next multiply these matrices by constants that have 0's in all positions except 0,4,8,12, and in those selected positions they have either 1, X, or $X+1$. Below we denote these constants by $\mathsf{C_1}$, $\mathsf{C_X}$ and $\mathsf{C_{X+1}}$, respectively. Setting

$$B'_0 = A\cdot\mathsf{C_X} + (A_1+A_6)\cdot\mathsf{C_1} + A_{11}\cdot\mathsf{C_{X+1}}, \qquad B'_1 = (A+A_1)\cdot\mathsf{C_1} + A_6\cdot\mathsf{C_{X+1}} + A_{11}\cdot\mathsf{C_X},$$
$$B'_2 = (A+A_{11})\cdot\mathsf{C_1} + A_1\cdot\mathsf{C_{X+1}} + A_6\cdot\mathsf{C_X}, \qquad B'_3 = A\cdot\mathsf{C_{X+1}} + A_1\cdot\mathsf{C_X} + (A_6+A_{11})\cdot\mathsf{C_1}$$

we get that the top rows of the four B'_i's contain the four rows of the resulting matrix B after mix-column, and moreover all the other rows in the B'_i's are zero. Having computed all the rows of the result, we use three more rotations to move them to place, namely set $B = B'_0 + (B'_1 \gg 1) + (B'_2 \gg 2) + (B'_3 \gg 3)$. A pseudo-code of the combined shift-row/mix-column operation is given below:

		Level	
Input: ciphertext $\mathfrak{c}''$		$t - 3.5$	
10.	$\mathfrak{c}''_j \leftarrow \mathfrak{c}'' \gg j$ for $j = 0, 1, 6, 11$	$t - 3.5$	// Rotations
11.	$\mathfrak{c}^*_0 \leftarrow \mathfrak{c}''_0 \cdot \mathsf{C}_{\mathsf{X}} + (\mathfrak{c}''_1 + \mathfrak{c}''_6)\mathsf{C}_1 + \mathfrak{c}''_{11} \cdot \mathsf{C}_{\mathsf{X}+1}$ $\mathfrak{c}^*_1 \leftarrow (\mathfrak{c}''_0 + \mathfrak{c}''_1)\mathsf{C}_1 + \mathfrak{c}''_6 \cdot \mathsf{C}_{\mathsf{X}+1} + \mathfrak{c}''_{11} \cdot \mathsf{C}_{\mathsf{X}}$ $\mathfrak{c}^*_2 \leftarrow (\mathfrak{c}''_0 + \mathfrak{c}''_{11})\mathsf{C}_1 + \mathfrak{c}''_1 \cdot \mathsf{C}_{\mathsf{X}+1} + \mathfrak{c}''_6 \cdot \mathsf{C}_{\mathsf{X}}$ $\mathfrak{c}^*_3 \leftarrow \mathfrak{c}''_0 \cdot \mathsf{C}_{\mathsf{X}+1} + \mathfrak{c}''_1 \cdot \mathsf{C}_{\mathsf{X}} + (\mathfrak{c}''_6 + \mathfrak{c}''_{11})\mathsf{C}_1$	$t - 4$	// Linear combinations
12.	Output $\mathfrak{c}^*_0 + (\mathfrak{c}^*_1 \gg 1) + (\mathfrak{c}^*_2 \gg 2) + (\mathfrak{c}^*_3 \gg 3)$	$t - 4$	// Assembling the result

4.1.3 The Cost of One Round Function

The above description yields an estimate of 4 levels for implementing one round function, which is indeed what we get in our experiments. The time complexity is dominated by the number of key-switching operations, which we need to do for every multiplication and every automorphism. The byte-substitution takes three multiplications and four automorphisms for inversion, and seven more automorphisms for the affine transformation, for a total of 14 key-switches. The shift-row/mix-column operation adds six more automorphisms, for a grand total of 20 key-switches per round.

We mention that the byte-slice implementation in Section 4.2 below would consume the same number of levels but use less key-switching operations per round since the shift-row/column-mix operation no longer needs automorphisms. Hence we would get 14 rather than 20 key-switching operations per round, so we expect the amortized complexity of this implementation to be faster by a factor of $20/14 \approx 1.4$. However, since we need to manipulate 16 times as many ciphertexts, the implementation would take much more time per evaluation (by a factor of $16 \cdot 14/20 = 11.2$) and require more memory.

4.2 Byte- and Bit-Slice Implementations

In the byte sliced implementation we use sixteen distinct ciphertexts to represent a single state matrix. (But since each ciphertext can hold ℓ plaintext slots, then these 16 ciphertexts can hold the state of ℓ different AES blocks). In this representation there is no interaction between the slots, thus we operate with pure ℓ-fold SIMD operations. The AddKey and SubBytes steps are exactly as above (except applied to 16 ciphertexts rather than a single one). The permutations in the ShiftRows/MixColumns step are now "for free", but the scalar multiplication in MixColumns still consumes 1/2 level in the modulus chain.

For the bit sliced implementation we represent the entire round function as a binary circuit, and we use 128 distinct ciphertexts (one per bit of the state matrix). However each set of 128 ciphertexts is able to represent a total of ℓ distinct blocks. The main issue here is how to create a circuit for the round function which is as shallow, in terms of number of multiplication gates, as possible. Again the main issue is the SubBytes operation as all operations are essentially linear. To implement the SubBytes we used the "depth-16" circuit of Boyar and Peralta [3], which consumes four levels. The rest of the round function can be represented as a set of bit-additions, Thus, implementing this method means that we should again consume only four levels per level.

4.3 Using Bootstrapping

Without bootstrapping, implementing ten rounds requires over 40 levels in the modulus chain, which means that we need a very large dimension to get security. We could hope to use the "bootstrapping as optimization" technique from BGV [5] to get smaller dimension, and hence speed up the computation. As it turns

Test	m	$\phi(m)$	lvls	$\|Q\|$	security	params/key-gen	Encrypt	Decrypt	memory
no bootstrap	53261	46080	40	886	150-bit	26.45 / 73.03	245.1	394.3	3GB
bootstrap	28679	23040	23	493	123-bit	148.2 / 37.2	1049.9	1630.5	3.7GB

Table 1: Performence results of homomorphic AES. Time is in seconds, the modulus size $|Q|$ includes extra primes as in Section 3.1.

out, however, the reduction in dimension is not enough to compensate for the extra time spent in the recryption procedure itself, so this does not lead to faster process. Bootstrapping is still needed, however, in applications that further process the result of the AES encryption. Hence in our implementation we also tested incorporating recryption into the AES computation.

One avenue for optimization in this case is to recrypt several ciphertexts together: The implementation of recryption in HElib handles "fully packed ciphertexts" whose slots contain elements from $\mathbb{F}_{2^d}$ (for some d divisible by 8), but our AES implementation only uses $\mathbb{F}_{2^8}$ elements (i.e. bytes) in the slots. We can therefore recrypt several ciphertexts together, packing $d/8$ bytes in each slot. Since in this setting most of the AES computation time is spent on recryption, we can process $d/8$ ciphertexts at nearly the same time as we do a single ciphertext, yielding a nearly $d/8$ speedup in amortized time. In our experiments we used $d = 24$, so this yields roughly a $3\times$ improvement.

4.4 Performance Details

As remarked in the introduction, we tested our implementations on a two-year-old Lenovo X230 laptop with Intel Core i5-3320M running at 2.6GHz, on an Ubuntu 14.04 VM with 4GB of RAM, using the g++ compiler version 4.9.2. The results of these tests are summarized in Table 1.

Non-bootstrapping implementation. For the non-bootstrapping experiment we selected parameters large enough to cope with 40 levels of computation. Appendix C contains our old derivation of the parameters to use, in our newer implementation we used instead the HElib derivation (that takes into consideration also the hybrid approach from Section 3.1), and is described in the HElib design document [18, Sec 3.1.4]. A rule-of-thumb is that for an L-level computation we need the dimension to be roughly $1000 \cdot L$. Specifically here we worked with the m-th cyclotomic for $m = 53261$, which yields lattices of dimension $\phi(m) = 46080$. This setting has 1920 slots, so we can fit $1920/16 = 120$ AES blocks in each ciphertext.

For this setting, key-generation took about 1.5 minutes, of which roughly 30 seconds were spent computing key-independent tables and about one minute was spent generating the keys and key-switching matrices. The input to the actual computation consisted of 120 plaintext blocks (in cleartext), and the eleven AES round keys encrypted in eleven packed ciphertext using our homomorphic encryption scheme. Homomorphic AES-encryption operation took 252 seconds, yielding throughput of 2 seconds per block.

Implementation using bootstrapping. Since bootstrapping in HElib takes about 12 levels, we chose our parameters here to cope with more than 20 levels of computation, so that we can compute at least two AES rounds per recryption. Specifically we had 23 computation levels and worked with $m = 28679$ and $\phi(m) = 23040$, a setting that yields 123-bit security by our estimates (see Equation (8) in Appendix C). This setting features 960 slots per ciphertext, each holding an element of $\mathbb{F}_{2^{24}}$, which is enough to pack 60 AES blocks.

Key-generation for this setting took about four minutes, three of which were spent computing key-independent tables, and under one minute spent on generating the keys and key-switching matrices. The input to the actual computation consisted of 180 plaintext blocks (in cleartext), and the same 11 packed cipehrtext encrypting the AES round keys. During the computation we applied the AES operation to three ciphertexts in parallel, and packed them into a single cipehrtext before each recryption.

The AES-encryption operation took 1050 seconds, of which 823 seconds were spent during two recryption operations, and the other 227 seconds were spent on the AES computation of the three ciphertexts. With 180 blocks, this gives throughput of 5.8 seconds per block. The entire computation used 3.7GB of memory.

Implementing AES decryption. We also implemented the AES decryption operation, basically by just reversing all the operations of the AES-encryption circuit. The operations performed in both cases are nearly identical (except a few multiply-by-constant operations), and yet in our tests the decryption time was about 60% slower than encryption.

For the non-bootstrapping case, one reason is that the AES encryption operation begins with inversion that lowers the level of the ciphertext, whereas decryption begins with the linear operations that keep the level more or less the same. As a result, operations on decryption are performed 2-3 levels higher than on encryption, which means that they need to manipulate more primes in our chain of moduli. It is not clear to us why this causes such a large slowdown, we speculate that some of it is the result of memory swapping or some other low-level effects.

For the bootstrapping case, the reason for the large slowdown is that the last inversion operation on decryption happens quite low in the chain, which triggers one more recryption operation, three on decryption vs. two on encryption. (This artifactc can probably be removed by special-casing the last round, but we did not attempt to do it.)

Acknowledgments

We thank Jean-Sebastien Coron for pointing out to us the efficient implementation from [27] of the AES S-box lookup.

References

[1] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 595–618. Springer, 2009.

[2] Sanjeev Arora and Rong Ge. New algorithms for learning in the presence of errors. In *ICALP*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011.

[3] Joan Boyar and René Peralta. A depth-16 circuit for the AES S-box. Manuscript, `http://eprint.iacr.org/2011/332`, 2011.

[4] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. Manuscript, `http://eprint.iacr.org/2012/078`, 2012.

[5] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science (ITCS'12)*, 2012. Available at `http://eprint.iacr.org/2011/277`.

[6] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS'11*. IEEE Computer Society, 2011.

[7] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[8] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 487–504. Springer, 2011.

[9] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 446–464. Springer, 2012.

[10] Ivan Damgård and Marcel Keller. Secure multiparty aes. In *Proc. of Financial Cryptography 2010*, volume 6052 of *LNCS*, pages 367–374, 2010.

[11] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. Manuscript, 2011.

[12] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 2008.

[13] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[14] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011.

[15] Craig Gentry, Shai Halevi, and Nigel Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012. Full version at `http://eprint.iacr.org/2011/566`.

[16] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013, Part I*, pages 75–92. Springer, 2013.

[17] Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Robustness of the learning with errors assumption. In *Innovations in Computer Science - ICS '10*, pages 230–240. Tsinghua University Press, 2010.

[18] Shai Halevi and Victor Shoup. Design and implementation of a homomorphic-encryption library. manuscript, available at `http://people.csail.mit.edu/shaih/pubs/he-library.pdf`, Accessed January 2015.

[19] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.

[20] C. Orlandi J.B. Nielsen, P.S. Nordholt and S. Sheshank. A new approach to practical active-secure two-party computation. Manuscript, 2011.

[21] Kristin Lauter, Michael Naehrig, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *CCSW*, pages 113–124. ACM, 2011.

[22] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, 2011.

[23] Adriana Lòpez-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *STOC*. ACM, 2012.

[24] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, 2010.

[25] Daniele Micciancio and Oded Regev. *Lattice-based cryptography*, pages 147–192. Springer, 2009.

[26] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Steven C. Williams. Secure two-party computation is practical. In *Proc. ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267, 2009.

[27] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.

[28] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography - PKC'10*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010.

[29] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. Manuscript at http://eprint.iacr.org/2011/133, 2011.

A More Details

Following [24, 5, 15, 29] we utilize rings defined by cyclotomic polynomials, $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$. We let $\mathbb{A}_q$ denote the set of elements of this ring reduced modulo various (possibly composite) moduli q. The ring $\mathbb{A}$ is the ring of integers of a the mth cyclotomic number field K.

A.1 Plaintext Slots

In our scheme plaintexts will be elements of $\mathbb{A}_2$, and the polynomial $\Phi_m(X)$ factors modulo 2 into ℓ irreducible factors, $\Phi_m(X) = F_1(X) \cdot F_2(X) \cdots F_\ell(X) \pmod 2$, all of degree $d = \phi(m)/\ell$. Just as in [5, 15, 29] each factor corresponds to a "plaintext slot". That is, we view a polynomial $a \in \mathbb{A}_2$ as representing an ℓ-vector $(a \bmod F_i)_{i=1}^{\ell}$.

It is standard fact that the Galois group $\mathcal{G}\mathsf{al} = \mathcal{G}\mathsf{al}(\mathbb{Q}(\zeta_m)/\mathbb{Q})$ consists of the mappings $\kappa_k : a(X) \mapsto a(x^k) \bmod \Phi_m(X)$ for all k co-prime with m, and that it is isomorphic to $(\mathbb{Z}/m\mathbb{Z})^*$. As noted in [15], for each $i, j \in \{1, 2, \ldots, \ell\}$ there is an element $\kappa_k \in \mathcal{G}\mathsf{al}$ which sends an element in slot i to an element in slot j. Namely, if $b = \kappa_i(a)$ then the element in the j'th slot of b is the same as that in the i'th slot of a. In addition $\mathcal{G}\mathsf{al}$ contains the Frobenius elements, $X \longrightarrow X^{2^i}$, which also act as Frobenius on the individual slots separately.

For the purpose of implementing AES we will be specifically interested in arithmetic in $\mathbb{F}_{2^8}$ (represented as $\mathbb{F}_{2^8} = \mathbb{F}_2[X]/G(X)$ with $G(X) = X^8 + X^4 + X^3 + X + 1$). We choose the parameters so that d is divisible by 8, so $\mathbb{F}_{2^d}$ includes $\mathbb{F}_{2^d}$ as a subfield. This lets us think of the plaintext space as containing ℓ-vectors over $\mathbb{F}_{2^n}$.

A.2 Canonical Embedding Norm

Following [24], we use as the "size" of a polynomial $a \in \mathbb{A}$ the l_∞ norm of its canonical embedding. Recall that the canonical embedding of $a \in \mathbb{A}$ into $\mathbb{C}^{\phi(m)}$ is the $\phi(m)$-vector of complex numbers $\sigma(a) = (a(\zeta_m^i))_i$ where ζ_m is a complex primitive m-th root of unity and the indexes i range over all of $(\mathbb{Z}/m\mathbb{Z})^*$. We call the norm of $\sigma(a)$ the *canonical embedding norm* of a, and denote it by

$$\|a\|_\infty^{\mathsf{can}} = \|\sigma(a)\|_\infty.$$

We will make use of the following properties of $\|\cdot\|_\infty^{\mathsf{can}}$:

- For all $a, b \in \mathbb{A}$ we have $\|a \cdot b\|_\infty^{\mathsf{can}} \leq \|a\|_\infty^{\mathsf{can}} \cdot \|b\|_\infty^{\mathsf{can}}$.
- For all $a \in \mathbb{A}$ we have $\|a\|_\infty^{\mathsf{can}} \leq \|a\|_1$.
- There is a ring constant c_m (depending only on m) such that $\|a\|_\infty \leq c_m \cdot \|a\|_\infty^{\mathsf{can}}$ for all $a \in \mathbb{A}$.

The ring constant c_m is defined by $c_m = \|\mathsf{CRT}_m^{-1}\|_\infty$ where CRT_m is the CRT matrix for m, i.e. the Vandermonde matrix over the complex primitive m-th roots of unity. Asymptotically the value c_m can grow super-polynomially with m, but for the "small" values of m one would use in practice values of c_m can be evaluated directly. See [11] for a discussion of c_m.

Canonical Reduction. When working with elements in $\mathbb{A}_q$ for some integer modulus q, we sometimes need a version of the canonical embedding norm that plays nice with reduction modulo q. Following [15], we define the *canonical embedding norm reduced modulo* q of an element $a \in \mathbb{A}$ as the smallest canonical embedding norm of any a' which is congruent to a modulo q. We denote it as

$$|a|_q^{can} \stackrel{\text{def}}{=} \min\{\, \|a'\|_\infty^{\mathsf{can}} : \, a' \in \mathbb{A}, \, a' \equiv a \pmod q \,\}.$$

We sometimes also denote the polynomial where the minimum is obtained by $[a]_q^{\mathsf{can}}$, and call it the *canonical reduction* of a modulo q. Neither the canonical embedding norm nor the canonical reduction is used in the scheme itself, it is only in the analysis of it that we will need them. We note that (trivially) we have $|a|_q^{\mathsf{can}} \leq \|a\|_\infty^{\mathsf{can}}$.

A.3 Double CRT Representation

As noted in Section 2, we usually represent an element $a \in A_q$ via double-CRT representation, with respect to both the polynomial factor of $\Phi_m(X)$ and the integer factors of q. Specifically, we assume that $\mathbb{Z}/q\mathbb{Z}$ contains a primitive m-th root of unity (call it ζ), so $\Phi_m(X)$ factors modulo q to linear terms $\Phi_m(X) = \prod_{i \in (\mathbb{Z}/m\mathbb{Z})^*}(X - \zeta^j) \pmod q$. We also denote q's prime factorization by $q = \prod_{i=0}^t p_i$. Then a polynomial $a \in \mathbb{A}_q$ is represented as the $(t+1) \times \phi(m)$ matrix of its evaluation at the roots of $\Phi_m(X)$ modulo p_i for $i = 0, \ldots, t$:

$$\mathsf{dble\text{-}CRT}^t(a) = \left(a\left(\zeta^j\right) \bmod p_i\right)_{0 \leq i \leq t, j \in (\mathbb{Z}/m\mathbb{Z})^*}.$$

The double CRT representation can be computed using $t+1$ invocations of the FFT algorithm modulo the p_i, picking only the FFT coefficients which correspond to elements in $(\mathbb{Z}/m\mathbb{Z})^*$. To invert this representation we invoke the inverse FFT algorithm $t+1$ times on a vector of length m consisting of the thinned out values padded with zeros, then apply the Chinese Remainder Theorem, and then reduce modulo $\Phi_m(X)$ and q.

Addition and multiplication in $\mathbb{A}_q$ can be computed as component-wise addition and multiplication of the entries in the two tables (modulo the appropriate primes p_i),

$$\begin{aligned}\mathsf{dble\text{-}CRT}^t(a+b) &= \mathsf{dble\text{-}CRT}^t(a) + \mathsf{dble\text{-}CRT}^t(b)\\ \mathsf{dble\text{-}CRT}^t(a\cdot b) &= \mathsf{dble\text{-}CRT}^t(a) \cdot \mathsf{dble\text{-}CRT}^t(b).\end{aligned}$$

Also, for an element of the Galois group $\kappa_k \in \mathcal{G}\mathsf{al}$ (which maps $a(X) \in \mathbb{A}$ to $a(X^k) \bmod \Phi_m(X)$), we can evaluate $\kappa_k(a)$ on the double-CRT representation of a just by permuting the columns in the matrix, sending each column j to column $j \cdot k \bmod m$.

A.4 Sampling From $\mathbb{A}_q$

At various points we will need to sample from $\mathbb{A}_q$ with different distributions, as described below. We denote choosing the element $a \in \mathbb{A}$ according to distribution $\mathcal{D}$ by $a \leftarrow \mathcal{D}$. The distributions below are described as over $\phi(m)$-vectors, but we always consider them as distributions over the ring $\mathbb{A}$, by identifying a polynomial $a \in \mathbb{A}$ with its coefficient vector.

The uniform distribution $\mathcal{U}_q$: This is just the uniform distribution over $(\mathbb{Z}/q\mathbb{Z})^{\phi(m)}$, which we identify with $(\mathbb{Z} \cap (-q/2, q/2])^{\phi(m)})$. Note that it is easy to sample from $\mathcal{U}_q$ directly in double-CRT representation.

The "discrete Gaussian" $\mathcal{DG}_q(\sigma^2)$: Let $\mathcal{N}(0,\sigma^2)$ denote the normal (Gaussian) distribution on real numbers with zero-mean and variance σ^2, we use drawing from $\mathcal{N}(0,\sigma^2)$ and rounding to the nearest integer as an approximation to the discrete Gaussian distribution. Namely, the distribution $\mathcal{DG}_{q_t}(\sigma^2)$ draws a real ϕ-vector according to $\mathcal{N}(0,\sigma^2)^{\phi(m)}$, rounds it to the nearest integer vector, and outputs that integer vector reduced modulo q (into the interval $(-q/2, q/2]$).

Sampling small polynomials, $\mathcal{ZO}(p)$ and $\mathcal{HWT}(h)$: These distributions produce vectors in $\{0,\pm 1\}^{\phi(m)}$.

For a real parameter $\rho \in [0,1]$, $\mathcal{ZO}(p)$ draws each entry in the vector from $\{0,\pm 1\}$, with probability $\rho/2$ for each of -1 and $+1$, and probability of being zero $1-\rho$.

For an integer parameter $h \le \phi(m)$, the distribution $\mathcal{HWT}(h)$ chooses a vector uniformly at random from $\{0,\pm 1\}^{\phi(m)}$, subject to the conditions that it has exactly h nonzero entries.

A.5 Canonical embedding norm of random polynomials

In the coming sections we will need to bound the canonical embedding norm of polynomials that are produced by the distributions above, as well as products of such polynomials. In some cases it is possible to analyze the norm rigorously using Chernoff and Hoeffding bounds, but to set the parameters of our scheme we instead use a heuristic approach that yields better constants:

Let $a \in \mathbb{A}$ be a polynomial that was chosen by one of the distributions above, hence all the (nonzero) coefficients in a are IID (independently identically distributed). For a complex primitive m-th root of unity ζ_m, the evaluation $a(\zeta_m)$ is the inner product between the coefficient vector of a and the fixed vector $\mathbf{z}_m = (1, \zeta_m, \zeta_m^2, \ldots)$, which has Euclidean norm exactly $\sqrt{\phi(m)}$. Hence the random variable $a(\zeta_m)$ has variance $V = \sigma^2\phi(m)$, where σ^2 is the variance of each coefficient of a. Specifically, when $a \leftarrow \mathcal{U}_q$ then each coefficient has variance $q^2/12$, so we get variance $V_U = q^2\phi(m)/12$. When $a \leftarrow \mathcal{DG}_q(\sigma^2)$ we get variance $V_G \approx \sigma^2\phi(m)$, and when $a \leftarrow \mathcal{ZO}(\rho)$ we get variance $V_Z = \rho\phi(m)$. When choosing $a \leftarrow \mathcal{HWT}(h)$ we get a variance of $V_H = h$ (but not $\phi(m)$, since a has only h nonzero coefficients).

Moreover, the random variable $a(\zeta_m)$ is a sum of many IID random variables, hence by the law of large numbers it is distributed similarly to a complex Gaussian random variable of the specified variance.[4] We therefore use $6\sqrt{V}$ (i.e. six standard deviations) as a high-probability bound on the size of $a(\zeta_m)$. Since the evaluation of a at all the roots of unity obeys the same bound, we use six standard deviations as our bound on the canonical embedding norm of a. (We chose six standard deviations since $\mathrm{erfc}(6) \approx 2^{-55}$, which is good enough for us even when using the union bound and multiplying it by $\phi(m) \approx 2^{16}$.)

In many cases we need to bound the canonical embedding norm of a product of two such "random polynomials". In this case our task is to bound the magnitude of the product of two random variables, both are distributed close to Gaussians, with variances σ_a^2, σ_b^2, respectively. For this case we use $16\sigma_a\sigma_b$ as our bound, since $\mathrm{erfc}(4) \approx 2^{-25}$, so the probability that both variables exceed their standard deviation by more than a factor of four is roughly 2^{-50}.

B The Basic Scheme

We now define our leveled HE scheme on L levels; including the Modulus-Switching and Key-Switching operations and the procedures for KeyGen, Enc, Dec, and for Add, Mult, Scalar-Mult, and Automorphism.

Recall that a ciphertext vector $\mathbf{c}$ in the cryptosystem is a valid encryption of $a \in \mathbb{A}$ with respect to secret key $\mathbf{s}$ and modulus q if $[[\langle \mathbf{c}, \mathbf{s}\rangle]_q]_2 = a$, where the inner product is over $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$, the operation $[\cdot]_q$ denotes modular reduction in coefficient representation into the interval $(-q/2, +q/2]$, and we require that the "noise" $[\langle \mathbf{c}, \mathbf{s}\rangle]_q$ is sufficiently small (in canonical embedding norm reduced mod q). In our implementation a "normal" ciphertext is a 2-vector $\mathbf{c} = (c_0, c_1)$, and a "normal" secret key is of the form $\mathbf{s} = (1, -\mathfrak{s})$, hence decryption takes the form

$$a \leftarrow [c_0 - c_1 \cdot \mathfrak{s}]_q \bmod 2. \tag{2}$$

B.1 Our Moduli Chain

We define the chain of moduli for our depth-L homomorphic evaluation by choosing L "small primes" $p_0, p_1, \ldots, p_{L-1}$ and the t'th modulus in our chain is defined as $q_t = \prod_{j=0}^{t} p_j$. (The sizes will be determined later.) The primes p_i's are chosen so that for all i, $\mathbb{Z}/p_i\mathbb{Z}$ contains a primitive m-th root of unity. Hence we can use our double-CRT representation for all $\mathbb{A}_{q_t}$.

This choice of moduli makes it easy to get a level-$(t-1)$ representation of $a \in \mathbb{A}$ from its level-t representation. Specifically, given the level-t double-CRT representation $\mathsf{dble\text{-}CRT}^t(a)$ for some $a \in \mathbb{A}_{q_t}$, we can simply remove from the matrix the row corresponding to the last small prime p_t, thus obtaining a level-$(t-1)$ representation of $a \bmod q_{t-1} \in \mathbb{A}_{q_{t-1}}$. Similarly we can get the double-CRT representation for lower levels by removing more rows. By a slight abuse of notation we write $\mathsf{dble\text{-}CRT}^{t'}(a) = \mathsf{dble\text{-}CRT}^t(a) \bmod q_{t'}$ for $t' < t$.

Recall that encryption produces ciphertext vectors valid with respect to the largest modulus q_{L-1} in our chain, and we obtain ciphertext vectors valid with respect to smaller moduli whenever we apply modulus-switching to decrease the noise magnitude. As described in Section 3.3, our implementation *dynamically* adjust levels, performing modulus switching when the dynamically-computed noise estimate becomes too large. Hence each ciphertext in our scheme is tagged with both its level t (pinpointing the modulus q_t relative to which this ciphertext is valid), and an estimate ν on the noise magnitude in this ciphertext. In other words,

[4] The mean of $a(\zeta_m)$ is zero, since the coefficients of a are chosen from a zero-mean distribution.

a ciphertext is a triple $(\mathbf{c}, t, \nu)$ with $0 \le t \le L-1$, $\mathbf{c}$ a vector over $\mathbb{A}_{q_t}$, and ν a real number which is used as our noise estimate.

B.2 Modulus Switching

The operation $\mathsf{SwitchModulus}(\mathfrak{c})$ takes the ciphertext $\mathfrak{c} = ((c_0, c_1), t, \nu)$ defined modulo q_t and produces a ciphertext $\mathfrak{c}' = ((c'_0, c'_1), t-1, \nu')$ defined modulo q_{t-1}, Such that $[c_0 - \mathfrak{s} \cdot c_1]_{q_t} \equiv [c'_0 - \mathfrak{s} \cdot c'_1]_{q_{t-1}} \pmod 2$, and ν' is smaller than ν. This procedure makes use of the function $\mathsf{Scale}(x, q, q')$ that takes an element $x \in \mathbb{A}_q$ and returns an element $y \in \mathbb{A}_{q'}$ such that in coefficient representation it holds that $y \equiv x \pmod 2$, and y is the closest element to $(q'/q) \cdot x$ that satisfies this mod-2 condition.

To maintain the noise estimate, the procedure uses the pre-set ring-constant c_m (cf. Appendix A.2) and also a pre-set constant B_{scale} which is meant to bound the magnitude of the added noise term from this operation. It works as follows:

$\mathsf{SwitchModulus}((c_0, c_1), t, \nu)$:

1. If $t < 1$ then abort; // Sanity check
2. $\nu' \leftarrow \frac{q_{t-1}}{q_t} \cdot \nu + B_{\mathsf{scale}}$; // Scale down the noise estimate
3. If $\nu' > q_{t-1}/2c_m$ then abort; // Another sanity check
4. $c'_i \leftarrow \mathsf{Scale}(c_i, q_t, q_{t-1})$ for $i = 0, 1$; // Scale down the vector
5. Output $((c'_0, c'_1), t-1, \nu')$.

The constant B_{scale} is set as $B_{\mathsf{scale}} = 2\sqrt{\phi(m)/3} \cdot (8\sqrt{h} + 3)$, where h is the Hamming weight of the secret key. (In our implementation we use $h = 64$, so we get $B_{\mathsf{scale}} \approx 77\sqrt{\phi(m)}$.) To justify this choice, we apply to the proof of the modulus switching lemma from [15, Lemma 13] (in the full version), relative to the canonical embedding norm. In that proof it is shown that when the noise magnitude in the input ciphertext $\mathbf{c} = (c_0, c_1)$ is bounded by ν, then the noise magnitude in the output vector $\mathbf{c}' = (c'_0, c'_1)$ is bounded by $\nu' = \frac{q_{t-1}}{q_t} \cdot \nu + \| \langle \mathbf{s}, \tau \rangle \|_\infty^{\mathsf{can}}$, provided that the last quantity is smaller than $q_{t-1}/2$.

Above τ is the "rounding error" vector, namely $\tau \stackrel{\text{def}}{=} (\tau_0, \tau_1) = (c'_0, c'_1) - \frac{q_{t-1}}{q_t}(c_0, c_1)$. Heuristically assuming that τ behaves as if its coefficients are chosen uniformly in $[-1, +1]$, the evaluation $\tau_i(\zeta)$ at an m-th root of unity ζ_m is distributed close to a Gaussian complex with variance $\phi(m)/3$. Also, $\mathfrak{s}$ was drawn from $\mathcal{HWT}(h)$ so $\mathfrak{s}(\zeta_m)$ is distributed close to a Gaussian complex with variance h. Hence we expect $\tau_1(\zeta)\mathfrak{s}(\zeta)$ to have magnitude at most $16\sqrt{\phi(m)/3 \cdot h}$ (recall that we use $h = 64$). We can similarly bound $\tau_0(\zeta_m)$ by $6\sqrt{\phi(m)/3}$, and therefore the evaluation of $\langle \mathbf{s}, \tau \rangle$ at ζ_m is bounded in magnitude (whp) by:

$$16\sqrt{\phi(m)/3 \cdot h} + 6\sqrt{\phi(m)/3} = 2\sqrt{\phi(m)/3} \cdot \left(8\sqrt{h} + 3\right) \approx 77\sqrt{\phi(m)} = B_{\mathsf{scale}} \tag{3}$$

B.3 Key Switching

After some homomorphic evaluation operations we have on our hands not a "normal" ciphertext which is valid relative to "normal" secret key, but rather an "extended ciphertext" $((d_0, d_1, d_2), q_t, \nu)$ which is valid with respect to an "extended secret key" $\mathbf{s}' = (1, -\mathfrak{s}, -\mathfrak{s}')$. Namely, this ciphertext encrypts the plaintext $a \in \mathbb{A}$ via

$$a = \left[\left[d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}' \cdot d_2\right]_{q_t}\right]_2$$

and the magnitude of the noise $\left[d_0 - \mathfrak{s} \cdot d_1 - d_2 \cdot \mathfrak{s}'\right]_{q_t}$ is bounded by ν. In our implementation, the component $\mathfrak{s}$ is always the same element $\mathfrak{s} \in \mathbb{A}$ that was drawn from $\mathcal{HWT}(h)$ during key generation, but $\mathfrak{s}'$ can vary depending on the operation. (See the description of multiplication and automorphisms below.)

To enable that translation, we use some "key switching matrices" that are included in the public key. (In our implementation these "matrices" have dimension 2×1, i.e., the consist of only two elements from $\mathbb{A}$.) As explained in Section 3.1, we save on space and time by artificially "boosting" the modulus we use from q_t up to $P \cdot q_t$ for some "large" modulus P. We note that in order to represent elements in $\mathbb{A}_{Pq_t}$ using our dble-CRT representation we need to choose P so that $\mathbb{Z}/P\mathbb{Z}$ also has primitive m-th roots of unity. (In fact in our implementation we pick P to be a prime.)

The key-switching "matrix". Denote by $Q = P \cdot q_{L-2}$ the largest modulus relative to which we need to generate key-switching matrices. To generate the key-switching matrix from $\mathbf{s}' = (1, -\mathfrak{s}, -\mathfrak{s}')$ to $\mathbf{s} = (1, -\mathfrak{s})$ (note that both keys share the same element $\mathfrak{s}$), we choose two element, one uniform and the other from our "discrete Gaussian",

$$a_{\mathfrak{s},\mathfrak{s}'} \leftarrow \mathcal{U}_Q \text{ and } e_{\mathfrak{s},\mathfrak{s}'} \leftarrow \mathcal{DG}_Q(\sigma^2),$$

where the variance σ is a global parameter (that we later set as $\sigma = 3.2$). The "key switching matrix" then consists of the single column vector

$$W[\mathfrak{s}' \to \mathfrak{s}] = \begin{pmatrix} b_{\mathfrak{s},\mathfrak{s}'} \\ a_{\mathfrak{s},\mathfrak{s}'} \end{pmatrix}, \text{ where } b_{\mathfrak{s},\mathfrak{s}'} \stackrel{\text{def}}{=} \left[\mathfrak{s} \cdot a_{\mathfrak{s},\mathfrak{s}'} + 2e_{\mathfrak{s},\mathfrak{s}'} + P\mathfrak{s}'\right]_Q. \tag{4}$$

Note that W above is defined modulo $Q = Pq_{L-2}$, but we need to use it relative to $Q_t = Pq_t$ for whatever the current level t is. Hence before applying the key switching procedure at level t, we reduce W modulo Q_t to get $W_t \stackrel{\text{def}}{=} [W]_{Q_t}$. It is important to note that since Q_t divides Q then W_t is indeed a key-switching matrix. Namely it is of the form $(b, a)^T$ with $a \in \mathcal{U}_{Q_t}$ and $b = [\mathfrak{s} \cdot a + 2e_{\mathfrak{s},\mathfrak{s}'} + P\mathfrak{s}']_{Q_t}$ (with respect to the same element $e_{\mathfrak{s},\mathfrak{s}'} \in \mathbb{A}$ from above).

The SwitchKey **procedure.** Given the extended ciphertext $\mathfrak{c} = ((d_0, d_1, d_2), t, \nu)$ and the key-switching matrix $W_t = (b, a)^T$, the procedure $\mathsf{SwitchKey}_{W_t}(\mathfrak{c})$ proceeds as follows:[5]

$\mathsf{SwitchKey}_{(b,a)}((d_0, d_1, d_2), t, \nu)$:

1. Set $\begin{pmatrix} c'_0 \\ c'_1 \end{pmatrix} \leftarrow \left[\begin{pmatrix} Pd_0 & b \\ Pd_1 & a \end{pmatrix}\begin{pmatrix} 1 \\ d_2 \end{pmatrix}\right]_{Q_t}$; // The actual key-switching operation
2. $c''_i \leftarrow \mathsf{Scale}(c'_i, Q_t, q_t)$ for $i = 0, 1$; // Scale the vector back down to q_t
3. $\nu' \leftarrow \nu + B_{\mathsf{Ks}} \cdot q_t/P + B_{\mathsf{scale}}$; // The constant B_{Ks} is determined below
4. Output $((c''_0, c''_1), t, \nu')$.

To argue correctness, observe that although the "actual key switching operation" from above looks superficially different from the standard key-switching operation $\mathbf{c}' \leftarrow W \cdot \mathbf{c}$, it is merely an optimization that takes advantage of the fact that both vectors $\mathbf{s}'$ and $\mathbf{s}$ share the element $\mathfrak{s}$. Indeed, we have the equality over $\mathbb{A}_{Q_t}$:

$$\begin{aligned} c'_0 - \mathfrak{s} \cdot c'_1 &= [(P \cdot d_0) + d_2 \cdot b_{\mathfrak{s},\mathfrak{s}'} - \mathfrak{s} \cdot \left((P \cdot d_1) + d_2 \cdot a_{\mathfrak{s},\mathfrak{s}'}\right) \\ &= P \cdot (d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}' d_2) + 2 \cdot d_2 \cdot \epsilon_{\mathfrak{s},\mathfrak{s}'}, \end{aligned}$$

[5]For simplicity we describe the SwitchKey procedure as if it always switches back to mod-q_t, but in reality if the noise estimate is large enough then it can switch directly to q_{t-1} instead.

so as long as both sides are smaller than Q_t we have the same equality also over $\mathbb{A}$ (without the mod-Q_t reduction), which means that we get

$$[c'_0 - \mathfrak{s} \cdot c'_1]_{Q_t} \;=\; [P \cdot (d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}' d_2) + 2 \cdot d_2 \cdot \epsilon_{\mathfrak{s},\mathfrak{s}'}]_{Q_t} \;\equiv\; [d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}' d_2]_{Q_t} \pmod 2.$$

To analyze the size of the added term $2d_2\epsilon_{\mathfrak{s},\mathfrak{s}'}$, we can assume heuristically that d_2 behaves like a uniform polynomial drawn from $\mathcal{U}_{q_t}$, hence $d_2(\zeta_m)$ for a complex root of unity ζ_m is distributed close to a complex Gaussian with variance $q_t^2\phi(m)/12$. Similarly $\epsilon_{\mathfrak{s},\mathfrak{s}'}(\zeta_m)$ is distributed close to a complex Gaussian with variance $\sigma^2\phi(m)$, so $2d_2(\zeta)\epsilon(\zeta)$ can be modeled as a product of two Gaussians, and we expect that with overwhelming probability it remains smaller than $2 \cdot 16 \cdot \sqrt{q_t^2\phi(m)/12 \cdot \sigma^2\phi(m)} \;=\; \frac{16}{\sqrt{3}} \cdot \sigma q_t \phi(m)$. This yields a heuristic bound $16/\sqrt{3} \cdot \sigma\phi(m) \cdot q_t \;=\; B_{\mathsf{Ks}} \cdot q_t$ on the canonical embedding norm of the added noise term, and if the total noise magnitude does not exceed $Q_t/2c_m$ then also in coefficient representation everything remains below $Q_t/2$. Thus our constant B_{Ks} is set as

$$\frac{16\sigma\phi(m)}{\sqrt{3}} \;\approx\; 9\sigma\phi(m) \;=\; B_{\mathsf{Ks}} \tag{5}$$

Finally, dividing by P (which is the effect of the Scale operation), we obtain the final ciphertext that we require, and the noise magnitude is divided by P (except for the added B_{scale} term).

B.4 Key-Generation, Encryption, and Decryption

The procedures below depend on many parameters, h, σ, m, the primes p_i and P, etc. These parameters will be determined later.

KeyGen(): Given the parameters, the key generation procedure chooses a low-weight secret key and then generates an LWE instance relative to that secret key. Namely, we choose

$$\mathfrak{s} \leftarrow \mathcal{HWT}(h), \quad a \leftarrow \mathcal{U}_{q_{L-1}}, \quad \text{and } e \leftarrow \mathcal{DG}_{q_{L-1}}(\sigma^2)$$

Then sets the secret key as $\mathfrak{s}$ and the public key as (a, b) where $b = [a \cdot s + 2e]_{q_{L-1}}$.

In addition, the key generation procedure adds to the public key some key-switching "matrices", as described in Appendix B.3. Specifically the matrix $W[\mathfrak{s}^2 \to \mathfrak{s}]$ for use in multiplication, and some matrices $W[\kappa_i(\mathfrak{s}) \to \mathfrak{s}]$ for use in automorphisms, for $\kappa_i \in \mathcal{G}\mathsf{al}$ whose indexes generates $(\mathbb{Z}/m\mathbb{Z})^*$ (including in particular κ_2).

$\mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m})$: To encrypt an element $m \in \mathbb{A}_2$, we choose one "small polynomial" (with $0, \pm 1$ coefficients) and two Gaussian polynomials (with variance σ^2),

$$v \leftarrow \mathcal{ZO}(0.5) \;\text{ and }\; e_0, e_1 \leftarrow \mathcal{DG}_{q_{L-1}}(\sigma^2)$$

Then we set $c_0 = b \cdot v + 2 \cdot e_0 + m,\; c_1 = a \cdot v + 2 \cdot e_1$, and set the initial ciphertext as $\mathfrak{c}' = (c_0, c_1, L-1, B_{\mathsf{clean}})$, where B_{clean} is a parameter that we determine below.

The noise magnitude in this ciphertext (B_{clean}) is a little larger than what we would like, so before we start computing on it we do one modulus-switch. That is, the encryption procedure sets $\mathfrak{c} \leftarrow \mathsf{SwitchModulus}(\mathfrak{c}')$ and outputs $\mathfrak{c}$. We can deduce a value for B_{clean} as follows:

$$\begin{aligned}
\left|c_0 - \mathfrak{s} \cdot c_1\right|_{q_t}^{\mathsf{can}} &\le \|c_0 - \mathfrak{s} \cdot c_1\|_\infty^{\mathsf{can}} \\
&= \|((a \cdot s + 2 \cdot e) \cdot v + 2 \cdot e_0 + \mathbf{m} - (a \cdot v + 2 \cdot e_1) \cdot \mathfrak{s}\|_\infty^{\mathsf{can}} \\
&= \|\mathbf{m} + 2 \cdot (e \cdot v + e_0 - e_1 \cdot \mathfrak{s})\|_\infty^{\mathsf{can}} \\
&\le \|\mathbf{m}\|_\infty^{\mathsf{can}} + 2 \cdot (\|e \cdot v\|_\infty^{\mathsf{can}} + \|e_0\|_\infty^{\mathsf{can}} + \|e_1 \cdot \mathfrak{s}\|_\infty^{\mathsf{can}})
\end{aligned}$$

Using our complex Gaussian heuristic from Appendix A.5, we can bound the canonical embedding norm of the randomized terms above by

$$\|e \cdot v\|_\infty^{\mathsf{can}} \ \le \ 16\sigma\phi(m)/\sqrt{2}, \quad \|e_0\|_\infty^{\mathsf{can}} \ \le \ 6\sigma\sqrt{\phi(m)}, \quad \|e_1 \cdot \mathfrak{s}\|_\infty^{\mathsf{can}} \ \le \ 16\sigma\sqrt{h \cdot \phi(m)}$$

Also, the norm of the input message m is clearly bounded by $\phi(m)$, hence (when we substitute our parameters $h = 64$ and $\sigma = 3.2$) we get the bound

$$\phi(m) + 32\sigma\phi(m)/\sqrt{2} + 12\sigma\sqrt{\phi(m)} + 32\sigma\sqrt{h \cdot \phi(m)} \ \approx \ 74\phi(m) + 858\sqrt{\phi(m)} \ = \ B_{\mathsf{clean}} \quad (6)$$

Our goal in the initial modulus switching from q_{L-1} to q_{L-2} is to reduce the noise from its initial level of $B_{\mathsf{clean}} = \Theta(\phi(m))$ to our base-line bound of $B = \Theta(\sqrt{\phi(m)})$ which is determined in Equation (12) below.

$\underline{\mathsf{Dec}_{\mathfrak{pk}}(\mathfrak{c})}$: Decryption of a ciphertext (c_0, c_1, t, ν) at level t is performed by setting $m' \leftarrow [c_0 - \mathfrak{s} \cdot c_1]_{q_t}$, then converting m' to coefficient representation and outputting $m' \bmod 2$. This procedure works when $c_m \cdot \nu < q_t/2$, so this procedure only applies when the constant c_m for the field $\mathbb{A}$ is known and relatively small (which as we mentioned above will be true for all practical parameters). Also, we must pick the smallest prime $q_0 = p_0$ large enough, as described in Appendix C.2.

B.5 Homomorphic Operations

$\underline{\mathsf{Add}(\mathfrak{c}, \mathfrak{c}')}$: Given two ciphertexts $\mathfrak{c} = ((c_0, c_1), t, \nu)$ and $\mathfrak{c}' = ((c'_0, c'_1), t', \nu')$, representing messages $\mathbf{m}, \mathbf{m}' \in \mathbb{A}_2$, this algorithm forms a ciphertext $\mathfrak{c}_a = ((a_0, a_1), t_a, \nu_a)$ which encrypts the message $\mathbf{m}_a = \mathbf{m} + \mathbf{m}'$.

If the two ciphertexts do not belong to the same level then we reduce the larger one modulo the smaller of the two moduli, thus bringing them to the same level. (This simple modular reduction works as long as the noise magnitude is smaller than the smaller of the two moduli, if this condition does not hold then we need to do modulus switching rather than simple modular reduction.) Once the two ciphertexts are at the same level (call it t''), we just add the two ciphertext vectors and two noise estimates to get

$$\mathfrak{c}_a \ = \ \left(\left([c_0 + c'_0]_{q_{t''}}, [c_1 + c'_1]_{q_{t''}}\right), \ t'', \nu + \nu'\right).$$

$\underline{\mathsf{Mult}(\mathfrak{c}, \mathfrak{c}')}$: Given two ciphertexts representing messages $\mathbf{m}, \mathbf{m}' \in \mathbb{A}_2$, this algorithm forms a ciphertext encrypts the message $\mathbf{m} \cdot \mathbf{m}'$.

We begin by ensuring that the noise magnitude in both ciphertexts is smaller than the pre-set constant B (which is our base-line bound and is determined inEquation (12) below), performing modulus-switching as needed to ensure this condition. Then we bring both ciphertexts to the same level by reducing modulo the smaller of the two moduli (if needed). Once both ciphertexts have small noise magnitude and the same level we form the extended ciphertext (essentially performing the tensor product of the two) and apply key-switching to get back a normal ciphertext. A pseudo-code description of this procedure is given below.

$\underline{\mathsf{Mult}(\mathfrak{c}, \mathfrak{c}')}$:

1. While $\nu(\mathfrak{c}) > B$ do $\mathfrak{c} \leftarrow \mathsf{SwitchModulus}(\mathfrak{c})$; // $\nu(\mathfrak{c})$ is the noise estimate in $\mathfrak{c}$
2. While $\nu(\mathfrak{c}') > B$ do $\mathfrak{c}' \leftarrow \mathsf{SwitchModulus}(\mathfrak{c}')$; // $\nu(\mathfrak{c}')$ is the noise estimate in $\mathfrak{c}'$
3. Bring $\mathfrak{c}, \mathfrak{c}'$ to the same level t by reducing modulo the smaller of the two moduli
   Denote after modular reduction $\mathfrak{c} = ((c_0, c_1), t, \nu)$ and $\mathfrak{c}' = ((c'_0, c'_1), t, \nu')$

4. Set $(d_0, d_1, d_2) \leftarrow (c_0 \cdot c'_0\ ,\quad c_1 \cdot c'_0 + c_0 \cdot c'_1\ ,\quad -c_1 \cdot c'_1)$;
   Denote $\mathfrak{c}'' = ((d_0, d_1, d_2), t, \nu \cdot \nu')$
5. Output $\mathsf{SwitchKey}_{W[\mathfrak{s}^2 \to \mathfrak{s}]}(\mathfrak{c}'')$ // Convert to "normal" ciphertext

We stress that *the only place* where we force modulus switching is before the multiplication operation. In all other operations we allow the noise to grow, and it will be reduced back the first time it is input to a multiplication operation. We also note that we may need to apply modulus switching more than once before the noise is small enough.

Scalar-Mult$(\mathfrak{c}, \alpha)$: Given a ciphertext $\mathfrak{c} = (c_0, c_1, t, \nu)$ representing the message $\mathbf{m}$, and an element $\alpha \in \mathbb{A}_2$ (represented as a polynomial modulo 2 with coefficients in $\{-1, 0, 1\}$), this algorithm forms a ciphertext $\mathfrak{c}_m = (a_0, a_1, t_m, \nu_m)$ which encrypts the message $\mathbf{m}_m = \alpha \cdot \mathbf{m}$. This procedure is needed in our implementation of homomorphic AES, and is of more general interest in general computation over finite fields.

The algorithm makes use of a procedure $\mathsf{Randomize}(\alpha)$ which takes α and replaces each non-zero coefficients with a coefficients chosen at random from $\{-1, 1\}$. To multiply by α, we set $\beta \leftarrow \mathsf{Randomize}(\alpha)$ and then just multiply both c_0 and c_1 by β. Using the same argument as we used in Appendix A.5 for the distribution $\mathcal{HWT}(h)$, here too we can bound the norm of β by $\|\beta\|_\infty^{\mathsf{can}} \le 6\sqrt{\mathsf{Wt}(\alpha)}$ where $\mathsf{Wt}(\alpha)$ is the number of nonzero coefficients of α. Hence we multiply the noise estimate by $6\sqrt{\mathsf{Wt}(\alpha)}$, and output the resulting ciphertext $\mathfrak{c}_m = (c_0 \cdot \beta,\ c_1 \cdot \beta,\ t,\ \nu \cdot 6\sqrt{\mathsf{Wt}(\alpha)})$.

Automorphism$(\mathfrak{c}, \kappa)$: In the main body we explained how permutations on the plaintext slots can be realized via using elements $\kappa \in \mathcal{G}\mathsf{al}$; we also require the application of such automorphism to implement the Frobenius maps in our AES implementation.

For each κ that we want to use, we need to include in the public key the "matrix" $W[\kappa(\mathfrak{s}) \to \mathfrak{s}]$. Then, given a ciphertext $\mathfrak{c} = (c_0, c_1, t, \nu)$ representing the message $\mathbf{m}$, the function $\mathsf{Automorphism}(\mathfrak{c}, \kappa)$ produces a ciphertext $\mathfrak{c}' = (c'_0, c'_1, t, \nu')$ which represents the message $\kappa(\mathbf{m})$. We first set an "extended ciphertext" by setting

$$d_0 = \kappa(c_0), \quad d_1 \leftarrow 0, \quad \text{and } d_2 \leftarrow \kappa(c_1)$$

and then apply key switching to the extended ciphertext $((d_0, d_1, d_2), t, \nu)$ using the "matrix" $W[\kappa(\mathfrak{s}) \to \mathfrak{s}]$.

C Security Analysis and Parameter Settings

Below we derive the concrete parameters for use in our early implementation. This part of the report is outdated, we left it here for historical purpose.

We begin in Appendix C.1 by deriving a lower-bound on the dimension N of the LWE problem underlying our key-switching matrices, as a function of the modulus and the noise variance. (This will serve as a lower-bound on $\phi(m)$ for our choice of the ring polynomial $\Phi_m(X)$.) Then in Appendix C.2 we derive a lower bound on the size of the largest modulus Q in our implementation, in terms of the noise variance and the dimension N. Then in Appendix C.3 we choose a value for the noise variance (as small as possible subject to some nominal security concerns), solve the somewhat circular constraints on N and Q, and set all the other parameters.

C.1 Lower-Bounding the Dimension

Below we apply to the LWE-security analysis of Lindner and Peikert [22], together with a few (arguably justifiable) assumptions, to analyze the dimension needed for different security levels. The analysis below

assumes that we are given the modulus Q and noise variance σ^2 for the LWE problem (i.e., the noise is chosen from a discrete Gaussian distribution modulo Q with variance σ^2 in each coordinate). The goal is to derive a lower-bound on the dimension N required to get any given security level. The first assumption that we make, of course, is that the Lindner-Peikert analysis — which was done in the context of standard LWE — applies also for our ring-LWE case. We also make the following extra assumptions:

- We assume that (once σ is not too tiny), the security depends on the ratio Q/σ and not on Q and σ separately. Nearly all the attacks and hardness results in the literature support this assumption, with the exception of the Arora-Ge attack [2] (that works whenever σ is very small, regardless of Q).

- The analysis in [22] devised an experimental formula for the time that it takes to get a particular quality of reduced basis (i.e., the parameter δ of Gama and Nguyen [12]), then provided another formula for the advantage that the attack can derive from a reduced basis at a given quality, and finally used a computer program to solve these formulas for some given values of N and δ. This provides some time/advantage tradeoff, since obtaining a smaller value of δ (i.e., higher-quality basis) takes longer time and provides better advantage for the attacker.

  For our purposes we made the assumption that the best runtime/advantage ratio is achieved in the high-advantage regime. Namely we should spend basically all the attack running time doing lattice reduction, in order to get a good enough basis that will break security with advantage (say) 1/2. This assumption is consistent with the results that are reported in [22].

- Finally, we assume that to get advantage of close to 1/2 for an LWE instance with modulus Q and noise σ, we need to be able to reduce the basis well enough until the shortest vector is of size roughly Q/σ. Again, this is consistent with the results that are reported in [22].

Given these assumptions and the formulas from [22], we can now solve the dimension/security tradeoff analytically. Because of the first assumption we might as well simplify the equations and derive our lower bound on N for the case $\sigma = 1$, where the ratio Q/σ is equal to Q. (In reality we will use $\sigma \approx 4$ and increase the modulus by the same 2 bits).

Following Gama-Nguyen [12], recall that a reduced basis $B = (b_1|b_2|\dots|b_m)$ for a dimension-M, determinant-D lattice (with $\|b_1\| \le \|b_2\| \le \cdots \|b_M\|$), has quality parameter δ if the shortest vector in that basis has norm $\|b_1\| = \delta^M \cdot D^{1/M}$. In other words, the quality of B is defined as $\delta = \|b_1\|^{1/M}/D^{1/M^2}$. The time (in seconds) that it takes to compute a reduced basis of quality δ for a random LWE instance was estimated in [22] to be at least

$$\log(\text{time}) \;\ge\; 1.8/\log(\delta) - 110. \tag{7}$$

For a random Q-ary lattice of rank N, the determinant is exactly Q^N whp, and therefore a quality-δ basis has $\|b_1\| = \delta^M \cdot Q^{N/M}$. By our second assumption, we should reduce the basis enough so that $\|b_1\| = Q$, so we need $Q = \delta^M \cdot Q^{N/M}$. The LWE attacker gets to choose the dimension M, and the best choice for this attack is obtained when the right-hand-side of the last equality is minimized, namely for $M = \sqrt{N \log Q/\log\delta}$. This yields the condition

$$\log Q = \log(\delta^M Q^{N/M}) = M\log\delta + (N/M)\log Q = 2\sqrt{N \log Q \log\delta},$$

which we can solve for N to get $N = \log Q/4\log\delta$. Finally, we can use Equation (7) to express $\log\delta$ as a function of $\log(\text{time})$, thus getting $N = \log Q \cdot (\log(\text{time}) + 110)/7.2$. Recalling that in our case we used

$\sigma = 1$ (so $Q/\sigma = Q$), we get our lower-bound on N in terms of Q/σ. Namely, to ensure a time/advantage ratio of at least 2^k, we need to set the rank N to be at least

$$N \geq \frac{\log(Q/\sigma)(k+110)}{7.2} \tag{8}$$

For example, the above formula says that to get 80-bit security level we need to set $N \geq \log(Q/\sigma) \cdot 26.4$, for 100-bit security level we need $N \geq \log(Q/\sigma) \cdot 29.1$, and for 128-bit security level we need $N \geq \log(Q/\sigma) \cdot 33.1$. We comment that these values are indeed consistent with the values reported in [22].

C.1.1 LWE with Sparse Key

The analysis above applies to "generic" LWE instance, but in our case we use very sparse secret keys (with only $h = 64$ nonzero coefficients, all chosen as ± 1). This brings up the question of whether one can get better attacks against LWE instances with a very sparse secret (much smaller than even the noise). We note that Goldwasser et al. proved in [17] that LWE with low-entropy secret is as hard as standard LWE with weaker parameters (for large enough moduli). Although the specific parameters from that proof do not apply to our choice of parameter, it does indicate that weak-secret LWE is not "fundamentally weaker" than standard LWE. In terms of attacks, the only attack that we could find that takes advantage of this sparse key is by applying the reduction technique of Applebaum et al. [1] to switch the key with part of the error vector, thus getting a smaller LWE error.

In a sparse-secret LWE we are given a random N-by-M matrix A (modulo Q), and also an M-vector $\mathbf{y} = [\mathbf{s}A + \mathbf{e}]_Q$. Here the N-vector $\mathbf{s}$ is our very sparse secret, and $\mathbf{e}$ is the error M-vector (which is also short, but not sparse and not as short as $\mathbf{s}$).

Below let A_1 denotes the first N columns of A, A_2 the next N columns, then A_3, A_4, etc. Similarly $\mathbf{e}_1, \mathbf{e}_2, \ldots$ are the corresponding parts of the error vector and $\mathbf{y}_1, \mathbf{y}_2, \ldots$ the corresponding parts of $\mathbf{y}$. Assuming that A_1 is invertible (which happens with high probability), we can transform this into an LWE instance with respect to secret $\mathbf{e}_1$, as follows:

We have $\mathbf{y}_1 = \mathbf{s}A_1 + \mathbf{e}_1$, or alternatively $A_1^{-1}\mathbf{y}_1 = \mathbf{s} + A_1^{-1}\mathbf{e}_1$. Also, for $i > 1$ we have $\mathbf{y}_i = \mathbf{s}A_i + \mathbf{e}_i$, which together with the above gives $A_i A_1^{-1}\mathbf{y}_1 - \mathbf{y}_i = A_i A_1^{-1}\mathbf{e}_1 - \mathbf{e}_i$. Hence if we denote

$$B_1 \stackrel{\text{def}}{=} A_1^{-1}, \quad \text{and for } i > 1 \; B_i \stackrel{\text{def}}{=} A_i A 1^{-1},$$
$$\text{and similarly } \mathbf{z}_1 = A_1^{-1}\mathbf{y}_1, \quad \text{and for } i > 1 \;\; \mathbf{z}_i \stackrel{\text{def}}{=} A_i A_1^{-1}\mathbf{y}_i,$$

and then set $B \stackrel{\text{def}}{=} (B_1^t | B_2^t | B_3^t | \ldots)$ and $\mathbf{z} \stackrel{\text{def}}{=} (\mathbf{z}_1 | \mathbf{z}_2 | \mathbf{z}_3 | \ldots)$, and also $\mathbf{f} = (\mathbf{s} | \mathbf{e}_2 | \mathbf{e}_3 | \ldots)$ then we get the LWE instance

$$\mathbf{z} = \mathbf{e}_1^t B + \mathbf{f}$$

with secret $\mathbf{e}_1^t$. The thing that makes this LWE instance potentially easier than the original one is that the first part of the error vector f is our sparse/small vector s, so the transformed instance has smaller error than the original (which means that it is easier to solve).

Trying to quantify the effect of this attack, we note that the optimal M value in the attack from Appendix C.1 above is obtained at $M = 2N$, which means that the new error vector is $\mathbf{f} = (\mathbf{s}|\mathbf{e}_2)$, which has Euclidean norm smaller than $\mathbf{e} = (\mathbf{e}_1|\mathbf{e}_2)$ by roughly a factor of $\sqrt{2}$ (assuming that $\|\mathbf{s}\| \ll \|\mathbf{e}_1\| \approx \|\mathbf{e}_2\|$). Maybe some further improvement can be obtained by using a smaller value for M, where the shorter error may outweigh the "non optimal" value of M. However, we do not expect to get major improvement this way, so it seems that the very sparse secret should only add maybe one bit to the modulus/noise ratio.

C.2 The Modulus Size

In this section we assume that we are given the parameter $N = \phi(m)$ (for our polynomial ring modulo $\Phi_m(X)$). We also assume that we are given the noise variance σ^2, the number of levels in the modulus chain L, an additional "slackness parameter" ξ (whose purpose is explained below), and the number of nonzero coefficients in the secret key h. Our goal is to devise a lower bound on the size of the largest modulus Q used in the public key, so as to maintain the functionality of the scheme.

Controlling the Noise. Driving the analysis in this section is a bound on the noise magnitude right after modulus switching, which we denote below by B. We set our parameters so that starting from ciphertexts with noise magnitude B, we can perform one level of fan-in-two multiplications, then one level of fan-in-ξ additions, followed by key switching and modulus switching again, and get the noise magnitude back to the same B.

- Recall that in the "reduced canonical embedding norm", the noise magnitude is at most multiplied by modular multiplication and added by modular addition, hence after the multiplication and addition levels the noise magnitude grows from B to as much as ξB^2.
- As we've seen in Appendix B.3, performing key switching scales up the noise magnitude by a factor of P and adds another noise term of magnitude upto $B_{\mathsf{Ks}} \cdot q_t$ (before doing modulus switching to scale it back down). Hence starting from noise magnitude ξB^2, the noise grows to magnitude $P\xi B^2 + B_{\mathsf{Ks}} \cdot q_t$ (relative to the modulus Pq_t).

  Below we assume that after key-switching we do modulus switching directly to a smaller modulus.
- After key-switching we can switch to the next modulus q_{t-1} to decrease the noise back to our bound B. Following the analysis from Appendix B.2, switching moduli from Q_t to q_{t-1} decreases the noise magnitude by a factor of $q_{t-1}/Q_t = 1/(P \cdot p_t)$, and then add a noise term of magnitude B_{scale}.

  Starting from noise magnitude $P\xi B^2 + B_{\mathsf{Ks}} \cdot q_t$ before modulus switching, the noise magnitude after modulus switching is therefore bounded whp by

$$\frac{P \cdot \xi B^2 + B_{\mathsf{Ks}} \cdot q_t}{P \cdot p_t} + B_{\mathsf{scale}} \;=\; \frac{\xi B^2}{p_t} + \frac{B_{\mathsf{Ks}} \cdot q_{t-1}}{P} + B_{\mathsf{scale}}$$

Using the analysis above, our goal next is to set the parameters B, P and the p_t's (as functions of N, σ, L, ξ and h) so that in every level t we get $\frac{\xi B^2}{p_t} + \frac{B_{\mathsf{Ks}} \cdot q_{t-1}}{P} + B_{\mathsf{scale}} \le B$. Namely we need to satisfy at every level t the quadratic inequality (in B)

$$\frac{\xi}{p_t} B^2 \;-\; B \;+\; \Bigg(\underbrace{\frac{B_{\mathsf{Ks}} \cdot q_{t-1}}{P} + B_{\mathsf{scale}}}_{\text{denote this by } R_{t-1}} \Bigg) \;\le\; 0\,. \tag{9}$$

Observe that (assuming that all the primes p_t are roughly the same size), it suffices to satisfy this inequality for the largest modulus $t = L-2$, since R_{t-1} increases with larger t's. Noting that $R_{L-3} > B_{\mathsf{scale}}$, we want to get this term to be as close to B_{scale} as possible, which we can do by setting P large enough. Specifically, to make it as close as $R_{L-3} = (1 + 2^{-n})B_{\mathsf{scale}}$ it is sufficient to set

$$P \;\approx\; 2^n \frac{B_{\mathsf{Ks}} q_{L-3}}{B_{\mathsf{scale}}} \;\approx\; 2^n \frac{9\sigma N q_{L-3}}{77\sqrt{N}} \;\approx\; 2^{n-3} q_{L-3} \cdot \sigma\sqrt{N}, \tag{10}$$

Below we set (say) $n = 8$, which makes it close enough to use just $R_{L-3} \approx B_{\mathsf{scale}}$ for the derivation below.

Clearly to satisfy Inequality (9) we must have a positive discriminant, which means $1-4\frac{\xi}{p_{L-2}}R_{L-3} \geq 0$, or $p_{L-2} \geq 4\xi R_{L-3}$. Using the value $R_{L-3} \approx B_{\mathsf{scale}}$, this translates into setting

$$p_1 \approx p_2 \cdots \approx p_{L-2} \approx 4\xi \cdot B_{\mathsf{scale}} \approx 308\xi\sqrt{N} \tag{11}$$

Finally, with the discriminant positive and all the p_i's roughly the same size we can satisfy Inequality (9) by setting

$$B \approx \frac{1}{2\xi/p_{L-2}} = \frac{p_{L-2}}{2\xi} \approx 2B_{\mathsf{scale}} \approx 154\sqrt{N}. \tag{12}$$

The Smallest Modulus. After evaluating our L-level circuit, we arrive at the last modulus $q_0 = p_0$ with noise bounded by ξB^2. To be able to decrypt, we need this noise to be smaller than $q_0/2c_m$, where c_m is the ring constant for our polynomial ring modulo $\Phi_m(X)$. For our setting, that constant is always below 40, so a sufficient condition for being able to decrypt is to set

$$q_0 = p_0 \approx 80\xi B^2 \approx 2^{20.9}\xi N \tag{13}$$

The Encryption Modulus. Recall that freshly encrypted ciphertext have noise B_{clean} (as defined in Equation (6)), which is larger than our baseline bound B from above. To reduce the noise magnitude after the first modulus switching down to B, we therefore set the ratio $p_{L-1} = q_{L-1}/q_{L-2}$ so that $B_{\mathsf{clean}}/p_{L-1} + B_{\mathsf{scale}} \leq B$. This means that we set

$$p_{L-1} = \frac{B_{\mathsf{clean}}}{B - B_{\mathsf{scale}}} \approx \frac{74N + 858\sqrt{N}}{77\sqrt{N}} \approx \sqrt{N} + 11 \tag{14}$$

The Largest Modulus. Having set all the parameters, we are now ready to calculate the resulting bound on the largest modulus, namely $Q_{L-2} = q_{L-2} \cdot P$. Using Equations (11), and (13), we get

$$q_t = p_0 \cdot \prod_{i=1}^{t} p_i \approx (2^{20.9}\xi N) \cdot \left(308\xi\sqrt{N}\right)^t = 2^{20.9} \cdot 308^t \cdot \xi^{t+1} \cdot N^{t/2+1}. \tag{15}$$

Now using Equation (10) we have

$$\begin{aligned} P &\approx 2^5 q_{L-3}\sigma\sqrt{N} \approx 2^{25.9} \cdot 308^{L-3} \cdot \xi^{L-2} \cdot N^{(L-3)/2+1} \cdot \sigma\sqrt{N} \\ &\approx 2 \cdot 308^L \cdot \xi^{L-2}\sigma N^{L/2} \end{aligned}$$

and finally

$$\begin{aligned} Q_{L-2} = P \cdot q_{L-2} &\approx (2 \cdot 308^L \cdot \xi^{L-2}\sigma N^{L/2}) \cdot (2^{20.9} \cdot 308^{L-2} \cdot \xi^{L-1} \cdot N^{L/2}) \\ &\approx \sigma \cdot 2^{16.5L+5.4} \cdot \xi^{2L-3} \cdot N^L \end{aligned} \tag{16}$$

C.3 Putting It Together

We now have in Equation (8) a lower bound on N in terms of Q, σ and the security level k, and in Equation (16) a lower bound on Q with respect to N, σ and several other parameters. We note that σ is a free parameter, since it drops out when substituting Equation (16) in Equation (8). In our implementation we used $\sigma = 3.2$, which is the smallest value consistent with the analysis in [25].

For the other parameters, we set $\xi = 8$ (to get a small "wiggle room" without increasing the parameters much), and set the number of nonzero coefficients in the secret key at $h = 64$ (which is already included in the formulas from above, and should easily defeat exhaustive-search/birthday type of attacks). Substituting these values into the equations above we get

$$p_0 \approx 2^{23.9} N, \quad p_i \approx 2^{11.3}\sqrt{N} \text{ for } i = 1, \ldots, L-2$$
$$P \approx 2^{11.3L-5} N^{L/2}, \text{ and } \quad Q_{L-2} \approx 2^{22.5L-3.6} \sigma N^L.$$

Substituting the last value of Q_{L-2} into Equation (8) yields

$$N > \frac{(L(\log N + 23) - 8.5)(k + 110)}{7.2} \tag{17}$$

Targeting $k = 80$-bits of security and solving for several different depth parameters L, we get the results in the table below, which also lists approximate sizes for the primes p_i and P.

L	N	$\log_2(p_0)$	$\log_2(p_i)$	$\log_2(p_{L-1})$	$\log_2(P)$
10	9326	37.1	17.9	7.5	177.3
20	19434	38.1	18.4	8.1	368.8
30	29749	38.7	18.7	8.4	564.2
40	40199	39.2	18.9	8.6	762.2
50	50748	39.5	19.1	8.7	962.1
60	61376	39.8	19.2	8.9	1163.5
70	72071	40.0	19.3	9.0	1366.1
80	82823	40.2	19.4	9.1	1569.8
90	93623	40.4	19.5	9.2	1774.5

Choosing Concrete Values. Having obtained lower-bounds on $N = \phi(m)$ and other parameters, we now need to fix precise cyclotomic fields $\mathbb{Q}(\zeta_m)$ to support the algebraic operations we need. We have two situations we will be interested in for our experiments. The first corresponds to performing arithmetic on bytes in $\mathbb{F}_{2^8}$ (i.e. $n = 8$), whereas the latter corresponds to arithmetic on bits in $\mathbb{F}_2$ (i.e. $n = 1$). We therefore need to find an odd value of m, with $\phi(m) \approx N$ and m dividing $2^d - 1$, where we require that d is divisible by n. Values of m with a small number of prime factors are preferred as they give rise to smaller values of c_m. We also look for parameters which maximize the number of slots ℓ we can deal with in one go, and values for which $\phi(m)$ is close to the approximate value for N estimated above. When $n = 1$ we always select a set of parameters for which the ℓ value is at least as large as that obtained when $n = 8$.

	$n = 8$				$n = 1$			
L	m	$N = \phi(m)$	(d, ℓ)	c_K	m	$N = \phi(m)$	(d, ℓ)	c_K
10	11441	10752	(48,224)	3.60	11023	10800	(45,240)	5.13
20	34323	21504	(48,448)	6.93	34323	21504	(48,448)	6.93
30	31609	31104	(72,432)	5.15	32377	32376	(57,568)	1.27
40	54485	40960	(64,640)	12.40	42799	42336	(21,2016)	5.95
50	59527	51840	(72,720)	21.12	54161	52800	(60,880)	4.59
60	68561	62208	(72,864)	36.34	85865	63360	(60,1056)	12.61
70	82603	75264	(56,1344)	36.48	82603	75264	(56,1344)	36.48
80	92837	84672	(56,1512)	38.52	101437	85672	(42,2016)	19.13
90	124645	98304	(48,2048)	21.07	95281	94500	(45,2100)	6.22

D $\mathsf{Scale}(c, q_t, q_{t-1})$ in dble-CRT Representation

Let $q_i = \prod_{j=0}^{i} p_j$, where the p_j's are primes that split completely in our cyclotomic field $\mathbb{A}$. We are given a $c \in \mathbb{A}_{q_t}$ represented via double-CRT – that is, it is represented as a "matrix" of its evaluations at the primitive m-th roots of unity modulo the primes $p_0, \ldots, p_t$. We want to modulus switch to q_{t-1} – i.e., scale down by a factor of p_t. Let's recall what this means: we want to output $c' \in \mathbb{A}$, represented via double-CRT format (as its matrix of evaluations modulo the primes $p_0, \ldots, p_{t-1}$), such that

1. $c' = c \bmod 2$.
2. c' is very close (in terms of its coefficient vector) to c/p_t.

In the main body we explained how this could be performed in dble-CRT representation. This made explicit use of the fact that the two ciphertexts need to be equivalent modulo two. If we wished to replace two with a general prime p, then things are a bit more complicated. For completeness, although it is not required in our scheme, we present a methodology below. In this case, the conditions on $c^\dagger$ are as follows:

1. $c^\dagger = c \cdot p_t \bmod p$.
2. $c^\dagger$ is very close to c.
3. $c^\dagger$ is divisible by p_t.

As before, we set $c' \leftarrow c^\dagger / p_t$. (Note that for $p = 2$, we trivially have $c \cdot p_t = c \bmod p$, since p_t will be odd.)

This causes some complications, because we set $c^\dagger \leftarrow c + \delta$, where $\delta = -\bar{c} \bmod p_t$ (as before) but now $\delta = (p_t - 1) \cdot c \bmod p$. To compute such a δ, we need to know $c \bmod p$. Unfortunately, we don't have $c \bmod p$. One not-very-satisfying way of dealing with this problem is the following. Set $\hat{c} \leftarrow [p_t]_p \cdot c \bmod q_t$. Now, if c encrypted m, then $\hat{c}$ encrypts $[p_t]_p \cdot m$, and $\hat{c}$'s noise is $[p_t]_p < p/2$ times as large. It is obviously easy to compute $\hat{c}$'s double-CRT format from c's. Now, we set $c^\dagger$ so that the following is true:

1. $c^\dagger = \hat{c} \bmod p$.
2. $c^\dagger$ is very close to $\hat{c}$.
3. $c^\dagger$ is divisible by p_t.

This is easy to do. The algorithm to output $c^\dagger$ in double-CRT format is as follows:

1. Set $\bar{c}$ to be the coefficient representation of $\hat{c} \bmod p_t$. (Computing this requires a single "small FFT" modulo the prime p_t.)
2. Set δ to be the polynomial with coefficients in $(-p_t \cdot p/2, p_t \cdot p/2]$ such that $\delta = 0 \bmod p$ and $\delta = -\bar{c} \bmod p_t$.
3. Set $c^\dagger = \hat{c} + \delta$, and output $c^\dagger$'s double-CRT representation.
   (a) We already have $\hat{c}$'s double-CRT representation.
   (b) Computing δ's double-CRT representation requires t "small FFTs" modulo the p_j's.

E Other Optimizations

Some other optimizations that we encountered during our implementation work are discussed next. Not all of these optimizations are useful for our current implementation, but they may be useful in other contexts.

Three-way Multiplications. Sometime we need to multiply several ciphertexts together, and if their number is not a power of two then we do not have a complete binary tree of multiplications, which means that at some point in the process we will have three ciphertexts that we need to multiply together.

The standard way of implementing this 3-way multiplication is via two 2-argument multiplications, e.g., $x \cdot (y \cdot z)$. But it turns out that here it is better to use "raw multiplication" to multiply these three ciphertexts (as done in [7]), thus getting an "extended" ciphertext with four elements, then apply key-switching (and later modulus switching) to this ciphertext. This takes only six ring-multiplication operations (as opposed to eight according to the standard approach), three modulus switching (as opposed to four), and only one key switching (applied to this 4-element ciphertext) rather than two (which are applied to 3-element extended ciphertexts). All in all, this three-way multiplication takes roughly 1.5 times a standard two-element multiplication.

We stress that this technique is not useful for larger products, since for more than three multiplicands the noise begins to grow too large. But with only three multiplicands we get noise of roughly B^3 after the multiplication, which can be reduced to noise $\approx B$ by dropping two levels, and this is also what we get by using two standard two-element multiplications.

Commuting Automorphisms and Multiplications. Recalling that the automorphisms $X \mapsto X^i$ commute with the arithmetic operations, we note that some ordering of these operations can sometimes be better than others. For example, it may be better perform the multiplication-by-constant before the automorphism operation whenever possible. The reason is that if we perform the multiply-by-constant after the key-switching that follows the automorphism, then added noise term due to that key-switching is multiplied by the same constant, thereby making the noise slightly larger. We note that to move the multiplication-by-constant before the automorphism, we need to multiply by a different constant.

Switching to higher-level moduli. We note that it may be better to perform automorphisms at a higher level, in order to make the added noise term due to key-switching small with respect to the modulus. On the other hand operations at high levels are more expensive than the same operations at a lower level. A good rule of thumb is to perform the automorphism operations one level above the lowest one. Namely, if the naive strategy that never switches to higher-level moduli would perform some Frobenius operation at level q_i, then we perform the key-switching following this Frobenius operation at level Q_{i+1}, and then switch back to level q_{i+1} (rather then using Q_i and q_i).

Commuting Addition and Modulus-switching. When we need to add many terms that were obtained from earlier operations (and their subsequent key-switching), it may be better to first add all of these terms relative to the large modulus Q_i before switching the sum down to the smaller q_i (as opposed to switching all the terms individually to q_i and then adding).

Reducing the number of key-switching matrices. When using many different automorphisms $\kappa_i : X \mapsto X^i$ we need to keep many different key-switching matrices in the public key, one for every value of i that

we use. We can reduces this memory requirement, at the expense of taking longer to perform the automorphisms. We use the fact that the Galois group $\mathcal{G}\mathsf{al}$ that contains all the maps κ_i (which is isomorphic to $(\mathbb{Z}/m\mathbb{Z})^*$) is generated by a relatively small number of generators. (Specifically, for our choice of parameters the group $(\mathbb{Z}/m\mathbb{Z})^*$ has two or three generators.) It is therefore enough to store in the public key only the key-switching matrices corresponding to κ_{g_j}'s for these generators g_j of the group $\mathcal{G}\mathsf{al}$. Then in order to apply a map κ_i we express it as a product of the generators and apply these generators to get the effect of κ_i. (For example, if $i = g_1^2 \cdot g_2$ then we need to apply κ_{g_1} twice followed by a single application of κ_{g_2}.)

Better Bootstrapping in Fully Homomorphic Encryption

Craig Gentry[1], Shai Halevi[1], and Nigel P. Smart[2]

[1] IBM T.J. Watson Research Center
[2] Dept. Computer Science, University of Bristol

Abstract. Gentry's bootstrapping technique is currently the only known method of obtaining a "pure" fully homomorphic encryption (FHE) schemes, and it may offers performance advantages even in cases that do not require pure FHE (e.g., when using the noise-control technique of Brakerski-Gentry-Vaikuntanathan).
The main bottleneck in bootstrapping is the need to evaluate homomorphically the reduction of one integer modulo another. This is typically done by emulating a binary modular reduction circuit, using bit operations on binary representation of integers. We present a simpler approach that bypasses the homomorphic modular-reduction bottleneck to some extent, by working with a modulus very close to a power of two. Our method is easier to describe and implement than the generic binary circuit approach, and we expect it to be faster in practice (although we did not implement it yet). In some cases it also allows us to store the encryption of the secret key as a single ciphertext, thus reducing the size of the public key.
We also show how to combine our new method with the SIMD homomorphic computation techniques of Smart-Vercauteren and Gentry-Halevi-Smart, to get a bootstrapping method that works in time quasi-linear in the security parameter. This last part requires extending the techniques from prior work to handle arithmetic not only over fields, but also over some rings. (Specifically, our method uses arithmetic modulo a power of two, rather than over characteristic-two fields.)

Table of Contents

1 Introduction

Fully Homomorphic Encryption (FHE) [12, 7] is a powerful technique to enable a party to compute an arbitrary function on a set of encrypted inputs; and hence obtain the encryption of the function's output. Starting from Gentry's breakthrough result [6, 7], all known FHE schemes are constructed from *Somewhat Homomorphic Encryption* (SWHE) schemes, that can only evaluate functions of bounded complexity. The ciphertexts in these SWHE schemes include some "noise" to ensure security, and this noise grows when applying homomorphic operations until it becomes so large that it overwhelms the decryption algorithm and causes decryption errors. To overcome the growth of noise, Gentry used a *bootstrapping* transformation, where the decryption procedure is run homomorphically on a given ciphertext, using an encryption of the secret key that can be found in the public key,[3] resulting in a new ciphertext that encrypts the same message but has potentially smaller noise.

Over the last two years there has been a considerable amount of work on developing new constructions and optimizations [5, 13, 9, 3, 14, 2, 8, 1, 11], but all of these constructions still have noise that keeps growing and must be reduced before it overwhelms the decryption procedure. The techniques of Brakerski et al. [1] yield SWHE schemes where the noise grows slower, only linearly with the depth of the circuit being evaluated, but for any fixed public key one can still only evaluate circuits of fixed depth. The only known way to get "pure" FHE that can evaluate arbitrary functions with a fixed public key is by using bootstrapping. Also, bootstrapping can be used in conjunction with the techniques from [1] to get better parameters (and hence faster homomorphic evaluation), as described in [1, 11].

In nearly all SWHE schemes in the literature that support bootstrapping, decryption is computed by evaluating some ciphertext-dependent linear operation on the secret key, then reducing the result modulo a public odd modulus q into the range $(-q/2, q/2]$, and then taking the least significant bit of the result. Namely, denoting reduction modulo q by $[\cdot]_q$, we decrypt a ciphertext c by computing $a = [[L_c(s)]_q]_2$ where L_c is a linear function and s is the secret key. Given an encryption of the secret key s, computing an encryption of $L_c(s)$ is straightforward, and the bulk of the work in homomorphic decryption is devoted to reducing the result modulo q. This is usually done by computing encryptions of the bits in the binary representation of $L_c(s)$ and then emulating the binary circuit that reduces modulo q.

The starting point of this work is the observation that when q is very close to a power of two, the decryption formula takes a particularly simple form. Specifi-

[3] This transformation relies on the underlying SWHE being circularly secure.

cally, we can compute the linear function $L_c(s)$ modulo a power of two, and then XOR the top and bottom bits of the result. We then explain how to implement this simple decryption formula homomorphically, and also how the techniques of Gentry et al. from [11] can be used to compute this homomorphic decryption with only polylogarithmic overhead.

We note that applying the techniques from [11] to bootstrapping is not quite straightforward, because the input and output are not presented in the correct form for these techniques. (This holds both for the standard approach of emulating binary mod-q circuit and for our new approach.) Also, for our case we need to extend the results from [11] slightly, since we are computing a function over a ring (modulo a power of two) and not over a field.

We point out that in all work prior to [11], bootstrapping required adding to the public key many ciphertexts, encrypting the individual bits (or coefficients) of the secret key. This resulted in very large public keys, of size at least $\lambda^2 \cdot \text{polylog}(\lambda)$ (where λ is the security parameter). Using the techniques from [14, 1, 11], it is possible to encrypt the secret key in a "packed" form, hence reducing the number of ciphertexts to $O(\log \lambda)$ (so we can get public keys of size quasi-linear in λ). Using our technique from this work, it is even possible to store an encryption of the secret key as a single ciphertext, as described in Section 4. We next outline our main bootstrapping technique in a few more details.

Our method applies mainly to "leveled" schemes that use the noise control mechanism of Brakerski-Gentry-Vaikuntanathan [1].[4] Below and throughout this paper we concentrate on the BGV ring-LWE-based scheme, since it offers the most efficient homomorphic operations and the most room for optimizations.[5] The scheme is defined over a ring $R = \mathbb{Z}[X]/F(X)$ for a monic, irreducible polynomial $F(X)$ (over the integers $\mathbb{Z}$). For an arbitrary integer modulus n (not necessarily prime) we denote the ring $R_n \stackrel{\text{def}}{=} R/nR = (\mathbb{Z}/n\mathbb{Z})[X]/F(X)$. The scheme is parametrized by the number of levels that it can handle, which we denote by L, and by a set of decreasing odd moduli $q_0 \gg q_1 \gg \cdots \gg q_L$, one for each level.

The plaintext space is given by the ring R_2, while the ciphertext space for the i'th level consists of vectors in $(R_{q_i})^2$. Secret keys are polynomials $\mathfrak{s} \in R$ with "small" coefficients, and we view $\mathfrak{s}$ as the second element of the 2-vector $\mathbf{s} = (1, \mathfrak{s})$. A level-$i$ ciphertext $\mathbf{c} = (c_0, c_1)$ encrypts a plaintext polynomial $m \in R_2$ with respect to $\mathbf{s} = (1, \mathfrak{s})$ if we have the equality

[4] Our method can be used also with other schemes, as long as the scheme allows us to choose a modulus very close to a power of two. For example they can be used with the schemes from [3, 2].

[5] Our description of the BGV cryptosystem below assumes modulo-2 plaintext arithmetic, generalizing to modulo-p arithmetic for other primes $p > 2$ is straightforward.

over R, $[\langle \mathbf{c}, \mathbf{s} \rangle]_{q_i} = [c_0 + \mathfrak{s} \cdot c_1]_{q_i} \equiv m \pmod 2$, and moreover the polynomial $[c_0 + \mathfrak{s} \cdot c_1]_{q_i}$ is "small", i.e. all its coefficients are considerably smaller than q_i. Roughly, that polynomial is considered the "noise" in the ciphertext, and its coefficients grow as homomorphic operations are performed.[6] The crux of the noise-control technique from [1] is that a level-i ciphertext can be publicly converted into a level-$(i+1)$ ciphertext (with respect to the same secret key), and that this transformation reduces the noise in the ciphertext roughly by a factor of q_{i+1}/q_i.

Secret keys too are associated with levels, and the public key includes some additional information that (roughly speaking) makes it possible to convert a ciphertext with respect to level-i key $\mathbf{s}_i$ into a ciphertext with respect to level-$(i+1)$ key $\mathbf{s}_{i+1}$. In what follows we will only be interested in the secret keys at level L and level zero; which we will denote by $\mathbf{s}$ and $\tilde{\mathbf{s}}$ respectively to ease notation.

For bootstrapping, we have as input a level-L ciphertext (i.e. a vector $\mathbf{c} \in R/q_L R$ modulo the smallest modulus q_L). This means that the noise-control technique can no longer be applied to reduce the noise, hence (essentially) no more homomorphic operations can be performed on this ciphertext. To enable further computation, we must therefore "recrypt" the ciphertext $\mathbf{c}$, to obtain a new ciphertext that encrypts the same element of R with respect to some lower level $i < L$.

Our first observation is that the decryption at level L can be made more efficient when q_L is close to a power of two, specifically $q_L = 2^r + 1$ for an integer r, and moreover the coefficients of $Z = \langle \mathbf{c}, \mathbf{s} \rangle \bmod F(X)$ are much smaller than q_L^2 in magnitude. In particular if z is one of the coefficients of the polynomial Z then $[[z]_{q_L}]_2$ can be computed as $z\langle r \rangle \oplus z\langle 0 \rangle$, where $z\langle i \rangle$ is the i'th bit of z.

To evaluate the decryption formula homomorphically, we temporarily extend the plaintext space to polynomials modulo 2^{r+1} (rather than modulo 2). The level-L secret key is $\mathbf{s} = (1, \mathfrak{s})$, where all the coefficients of $\mathfrak{s}$ are small (in the interval $(-2^r, +2^r)$). We can therefore consider $\mathfrak{s}$ as a plaintext polynomial in $R/2^{r+1}R$, encrypt it inside a level-0 ciphertext, and keep that ciphertext in the public key. Thus, given the level-L ciphertext $\mathbf{c}$, we can evaluate the inner product $[\langle \mathbf{c}, \mathbf{s} \rangle \bmod F(X)]$ homomorphically, obtaining a level-0 ciphertext that encrypts the polynomial Z.

For simplicity, assume for now that what we get is an encryption of all the coefficients of Z separately. Given an encryption of a coefficient z of Z (which is an element in $\mathbb{Z}/2^{r+1}\mathbb{Z}$) we show in Section 3.1 how to extract (encryptions of) the zero'th and r'th bit using a data-oblivious algorithm. Hence we can fi-

[6] We ignore here the encryption procedure, since it does not play any role in the current work.

nally recover a new ciphertext, encrypting the same binary polynomial at a lower level $i < L$.

To achieve efficient bootstrapping, we exploit the ability to perform operations on elements modulo 2^{r+1} in a SIMD fashion (Single Instruction Multiple Data); much like in prior work [14, 1, 11]. Some care must be taken when applying these techniques in our case, since the inputs and outputs of the bootstrapping procedure are not in the correct format: Specifically, these techniques require that inputs and outputs be represented using polynomial Chinese Remainders (CRT representation), whereas decryption (and therefore recryption) inherently deals with polynomials in coefficient representation. We therefore must use explicit conversion to CRT representation, and ensure that these conversions are efficient enough. See details in Section 4.

Also, the techniques from prior work must be extended somewhat to be usable in our case: Prior work demonstrated that SIMD operations can be performed homomorphically when the underlying arithmetic is over a field, but in our case we have operations over the ring $\mathbb{Z}/2^{r+1}\mathbb{Z}$, which is not a field. The algebra needed to extend the SIMD techniques to this case is essentially an application of the theory of local fields [4]. We prove many of the basic results that we need in the full version [10], and refer the reader to [4] for a general introduction and more details.

Notations. Throughout the paper we denote by $[z]_q$ the reduction of z mod q into the interval $(-\frac{q}{2}, \frac{q}{2}]$. We also denote the i'th bit in the binary representation of the integer z by $z\langle i\rangle$. Similarly, when a is an integer polynomial of degree d with coefficients $(a_0, a_1, \ldots, a_d)$, we denote by $a\langle i\rangle$ the 0-1 degree-d polynomial whose coefficients are all the i'th bits $(a_0\langle i\rangle, a_1\langle i\rangle, \ldots, a_d\langle i\rangle)$. If $\mathbf{c}, \mathbf{s}$ are two same-dimension vectors, then $\langle \mathbf{c}, \mathbf{s}\rangle$ denotes their inner product.

Organization. We begin by presenting the simplified decryption formula in Section 2 and explain how to evaluate it homomorphically in Section 3. Then in Section 4 we recall some algebra and explain how to use techniques similar to [11] to run bootstrapping in time quasi-linear in the security parameter. Some of the proofs are omitted here, these are found in the full version of this work [10].

2 A simpler decryption formula

When the small modulus q_L has a special form – i.e. when it equals $u \cdot 2^r + v$ for some integer r and for some small positive odd integers u, v – then the mod-q_L decryption formula can be made to have a particularly simple form. Below we focus on the case of $q_L = 2^r + 1$, which suffices for our purposes.

So, assume that $q_L = 2^r + 1$ for some integer r and that we decrypt by setting $a \leftarrow [[\langle \mathbf{c}, \mathbf{s}\rangle \bmod F(X)]_{q_L}]_2$. Consider now the coefficients of the in-

teger polynomial $Z = \langle \mathbf{c}, \mathbf{s} \rangle \bmod F(X)$, without the reduction mod q_L. Since $\mathbf{s}$ has small coefficients (and we assume that reduction mod-$F(X)$ does not increase the coefficients by much) then all the coefficients of Z are much smaller than q_L^2. Consider one of these integer coefficients, denoted by z, so we know that $|z| \ll q_L^2 \approx 2^{2r}$. We consider the binary representation of z as a $2r$-bit integer, and assume for now that $z \geq 0$ and also $[z]_{q_L} \geq 0$. We claim that in this case, the bit $[[z]_{q_L}]_2$ can be computed simply as the sum of the lowest bit and the r'th bit of z, i.e., $[[z]_{q_L}]_2 = z\langle r \rangle \oplus z\langle 0 \rangle$. (Recall that $z\langle i \rangle$ is the i'th bit of z.)

Lemma 1. *Let $q = 2^r + 1$ for a positive integer r, and let z be a non-negative integer smaller than $\frac{q^2}{2} - q$, such that $[z]_q$ is also non-negative, $[z]_q \in [0, \frac{q}{2}]$. Then $[[z]_q]_2 = z\langle r \rangle \oplus z\langle 0 \rangle$.*

Proof. Let $z_0 = [z]_q \in [0, \frac{q}{2}]$, and consider the sequence of integers $z_i = z_0 + iq$ for $i = 0, 1, 2, \cdots$. Since we assume that $z \geq 0$ then z can be found in this sequence, say the k'th element $z = z_k = z_0 + kq$. Also since $z < \frac{q^2}{2} - q$ then $k = \lfloor z/q \rfloor < \frac{q}{2} - 1$. The bit that we want to compute is $[[z]_q]_2 = z_0\langle 0 \rangle$. We claim that $z_0\langle 0 \rangle = z_k\langle 0 \rangle + z_k\langle r \rangle \pmod 2$. This is because $z_k = z_0 + kq = z_0 + k(2^r + 1) = (z_0 + k) + k2^r$, which in particular means that $z_k\langle 0 \rangle = z_0\langle 0 \rangle + k\langle 0 \rangle \pmod 2$. But since $0 \leq z_0 \leq q/2$ and $0 \leq k < q/2 - 1$ then $0 \leq z_0 + k < q - 1 = 2^r$, so there is no carry bit from the addition $z_0 + k$ to the r'th bit position. It follows that the r'th bit of z_k is equal to the 0'th bit of k (i.e., $z_k\langle r \rangle = k\langle 0 \rangle$), and therefore $z_k\langle 0 \rangle = z_0\langle 0 \rangle + k\langle 0 \rangle = z_0\langle 0 \rangle + z_k\langle r \rangle \pmod 2$, which implies that $z_0\langle 0 \rangle = z_k\langle 0 \rangle + z_k\langle r \rangle \pmod 2$, as needed. □

We note that the proof can easily be extended for the case $q = u2^r + v$, if the bound on z is strengthened by a factor of v. To remove the assumption that both z and $[z]_q$ are non-negative, we use the following easy corollary:

Corollary 1. *Let $r \geq 3$ and $q = 2^r + 1$ and let z be an integer with absolute value smaller than $\frac{q^2}{4} - q$, such that $[z]_q \in (-\frac{q}{4}, \frac{q}{4})$. Then $[[z]_q]_2 = z\langle r \rangle \oplus z\langle r-1 \rangle \oplus z\langle 0 \rangle$.*

Proof. Denoting $z' = z + (q^2-1)/4 = z + (q+1)(q-1)/4 = \left(z + \frac{q-1}{4}\right) + q \cdot \frac{q-1}{4}$, we have $z' \equiv z + \frac{q-1}{4} \pmod q$ (since $\frac{q-1}{4} = 2^{r-2}$ is an integer). Moreover since $[z]_q \in (-\frac{q}{4}, \frac{q}{4}]$ then $[z]_q + \frac{q-1}{4} \in [0, q/2]$, hence $[z']_q = [z]_q + \frac{q-1}{4}$ (over the integers), and as $\frac{q-1}{4}$ is an even integer then $[z]_q = [z']_q \pmod 2$, or in other words $[[z]_q]_2 = [[z']_q]_2$. Since $z > -\frac{q^2}{4}$ and z is an integer then $z \geq -\frac{q^2-1}{4}$ and therefore $z' = z + \frac{q^2-1}{4} \geq 0$. Thus z' satisfies all the conditions set in Lemma 1, so applying that lemma we have $[[z]_q]_2 = [[z']_q]_2 = z'\langle r \rangle \oplus z'\langle 0 \rangle$.

We next observe that $z' = z + (q+1)(q-1)/4 = z + (2^r+2)2^{r-2} = z + 2^{r-1} + 2^{2r-2}$. Since $2r-2 > r$, this means that the bits 0 through r in the binary representation of z' are determined by $z + 2^{r-1}$ alone, so we have:

$$\begin{aligned} z'\langle i\rangle &= z\langle i\rangle \text{ for } i = 0, 1, \ldots, r-2 \\ z'\langle r-1\rangle &= 1 - z\langle r-1\rangle \\ z'\langle r\rangle &= \left\{ \begin{array}{ll} z\langle r\rangle & \text{if } z\langle r-1\rangle = 0 \\ 1 - z\langle r\rangle & \text{if } z\langle r-1\rangle = 1 \end{array} \right\} \;=\; z\langle r\rangle \oplus z\langle r-1\rangle \end{aligned}$$

Putting it all together, we get $[[z]_q]_2 = [[z']_q]_2 = z'\langle r\rangle \oplus z'\langle 0\rangle = z\langle r\rangle \oplus z\langle r-1\rangle \oplus z\langle 0\rangle$. □

Using Corollary 1 we can get our simplified decryption formula. First, we set our parameters such that $q_L = 2^r + 1$ and all the coefficients of the integer polynomial $Z = \langle \mathbf{c}, \mathbf{s}\rangle \bmod F(X)$ are smaller than $\frac{q_L^2}{4} - 1$ in absolute value, and moreover they are all less than $\frac{q_L-1}{4}$ away from a multiple of q_L. Given a two-element ciphertext $\mathbf{c} = (c_0, c_1) \in ((\mathbb{Z}/q_L\mathbb{Z})[X]/F(X))^2$, then compute $Z \leftarrow \langle \mathbf{c}, \mathbf{s}\rangle \bmod F(X)$ over the integers (without reduction mod q_L), and finally recover the plaintext as $Z\langle r\rangle + Z\langle r-1\rangle + Z\langle 0\rangle$. Ultimately, we obtain the plaintext polynomial $a \in \mathbb{F}_2[X]/F(X)$, where each coefficient in a is obtained as the XOR of bits 0, $r-1$, and r of the corresponding coefficient in Z.

Working modulo 2^{r+1}. Since we are only interested in the contents of bit positions 0, $r-1$, and r in the polynomial Z, we can compute Z modulo 2^{r+1} rather than over the integers. Observing that when $q_L = 2^r + 1$ then $\frac{q_L^2-1}{4} \equiv 2^{r-1} \pmod{2^{r+1}}$, our simplified decryption of a ciphertext vector $\mathbf{c} = (c_0, c_1)$ proceeds as follows:

1. Compute $Z \leftarrow [\langle \mathbf{c}, \mathbf{s}\rangle \bmod F(X)]_{2^{r+1}}$;
2. Recover the 0-1 plaintext polynomial $a = [Z\langle r\rangle + Z\langle r-1\rangle + Z\langle 0\rangle]_2$.

3 Basic Homomorphic Decryption

To get a homomorphic implementation of the simplified decryption formula from above, we use an instance of our homomorphic encryption scheme with underlying plaintext space $\mathbb{Z}_{2^{r+1}}$. Namely, denoting by $\tilde{\mathbf{s}}$ the level-0 secret-key and by q_0 the largest modulus, a ciphertext encrypting $a \in (\mathbb{Z}/2^{r+1}\mathbb{Z})[X]/F(X)$ with respect to $\tilde{\mathbf{s}}$ and q_0 is a 2-vector $\tilde{\mathbf{c}}$ over $(\mathbb{Z}/q_0\mathbb{Z})[X]/F(X)$ such that $|[\langle \tilde{\mathbf{c}}, \tilde{\mathbf{s}}\rangle \bmod F(X)]_{q_0}| \ll q_0$ and $[\langle \tilde{\mathbf{c}}, \tilde{\mathbf{s}}\rangle \bmod F(X)]_{q_0} \equiv a \pmod{2^{r+1}}$.

Recall that the ciphertext before bootstrapping is with respect to secret key $\mathbf{s}$ and modulus $q_L = 2^r + 1$. In this section we only handle the simple case where

the public key includes an encryption of each coefficient of the secret-key $\mathbf{s}$ separately. Namely, denoting $\mathbf{s} = (1, \mathfrak{s})$ and $\mathfrak{s}(X) = \sum_{j=0}^{d-1} \mathfrak{s}_j X^j$, we encode for each j the coefficient $\mathfrak{s}_j$ as the constant polynomial $\mathfrak{s}_j \in (\mathbb{Z}/2^{r+1}\mathbb{Z})[X]/F(X)$. (I.e., the degree-$d$ polynomial whose free term is $\mathfrak{s}_j \in [-2^r + 1, 2^r]$ and all the other coefficients are zero.) Then for each j we include in the public key a ciphertext $\tilde{\mathbf{c}}_j$ that encrypts this constant polynomial $\mathfrak{s}_j$ with respect to $\tilde{\mathbf{s}}$ and q_0. Below we abuse notations somewhat, using the same notation to refer both to a constant polynomial $z \in (\mathbb{Z}/2^r\mathbb{Z})[X]/F(X)$ and the free term of that polynomial $z \in (\mathbb{Z}/2^r\mathbb{Z})$.

Computing Z Homomorphically. Given the q_L-ciphertext $\mathbf{c} = (c_0, c_1)$ (that encrypts a plaintext polynomial $a \in \mathbb{F}_2[X]/F(X)$), we use the encryption of $\mathbf{s}$ from the public key to compute the simple decryption formula from above. Computing an encryption of $Z = [\langle \mathbf{c}, \mathbf{s} \rangle \bmod F(X)]_{2^{r+1}}$ is easy, since the coefficients of Z are just affine functions (over $(\mathbb{Z}/2^{r+1}\mathbb{Z})$) of the coefficients of $\mathfrak{s}$, which we can compute from the encryption of the $\mathfrak{s}_j$'s in the public key.

3.1 Extracting the Top and Bottom Bits

Now that we have encryptions of the coefficients of Z, we need to extract the relevant three bits in each of these coefficients and add them (modulo 2) to get encryptions of the plaintext coefficients. In more details, given a ciphertext $\tilde{\mathbf{c}}$ satisfying $[\langle \tilde{\mathbf{c}}, \tilde{\mathbf{s}} \rangle \bmod F(X)]_{q_0} \equiv z \pmod{2^{r+1}}$ where z is some constant polynomial, we would like to compute another ciphertext $\tilde{\mathbf{c}}$ satisfying $[\langle \tilde{\mathbf{c}}, \tilde{\mathbf{s}} \rangle \bmod F(X)]_{q_0} \equiv z\langle 0 \rangle + z\langle r-1 \rangle + z\langle r \rangle \pmod 2$ (with $[\langle \tilde{\mathbf{c}}, \tilde{\mathbf{s}} \rangle \bmod F(X)]_{q_0}$ still much smaller then q_0 in magnitude). To this end, we describe a procedure to compute for all $i = 0, 1, \ldots, r$ a ciphertext $\tilde{\mathbf{c}}_i$ satisfying $[\langle \tilde{\mathbf{c}}_i, \tilde{\mathbf{s}} \rangle \bmod F(X)]_{q_0} \equiv z\langle i \rangle \pmod 2$. Clearly, we can immediately set $\tilde{\mathbf{c}}_0 = \tilde{\mathbf{c}}$, we now describe how to compute the other $\tilde{\mathbf{c}}_i$'s.

The basic observation underlying this procedure is that modulo a power of 2, the second bit of $z - z^2$ is the same as that of z, but the LSB is zero-ed out. Thus setting $z' = (z - z^2)/2$ (which is an integer), we get that the LSB of z' is the second bit of z. More generally, we have the following lemma:

Lemma 2. *Let z be an integer with binary representation $z = \sum_{i=0}^{r} 2^i z\langle i \rangle$. Define $w_0 \stackrel{\text{def}}{=} z$, and for $i \geq 1$ define*

$$w_i \stackrel{\text{def}}{=} \frac{z - \sum_{j=0}^{i-1} 2^j w_j^{2^{i-j}} \bmod 2^{r+1}}{2^i} \quad \textit{(division by } 2^i \textit{ over the rationals).} \tag{1}$$

Then the w_i's are integers and we have $w_i\langle 0 \rangle = z\langle i \rangle$ for all i.

Proof. The lemma clearly holds for $i = 0$. Now fix some $i \geq 1$, assume that the lemma holds for all $j < i$, and we prove that it holds also for i. It is easy to show by induction that for any integer u and all $j \leq r$ we have

$$u^{2^j} \bmod 2^{r+1} = u\langle 0\rangle + 2^{j+1}t \text{ for some integer } t.$$

Namely, the LSB of $u^{2^j} \bmod 2^{r+1}$ is the same as the LSB of u, and the next j bits are all zero. This means that the bit representation of $v_j \stackrel{\text{def}}{=} 2^j w_j^{2^{i-j}} \bmod 2^{r+1}$ has bits $0, 1, \ldots, j-1$ all zero (due to the multiplication by 2^j), then $v_j\langle j\rangle = w_j\langle 0\rangle = z\langle j\rangle$ (by the induction hypothesis), and the next $i-j$ bits are again zero (by the observation above). In other words, the lowest $i+1$ bits of v_j are all zero, except the j'th bit which is equal to the j'th bit of z.

This means that the lowest i bits of the sum $\sum_{j=0}^{i-1} v_j$ are the same as the lowest i bits of z, and the $i+1$'st bit of the sum is zero. Hence the lowest i bits of $z - \sum_{j=0}^{i-1} v_j$ are all zero, and the $i+1$'st bit is $z\langle i\rangle$. Hence $z - \sum_{j=0}^{i-1} v_j$ is divisible by 2^i (over the integers), and the lowest bit of the result is $z\langle i\rangle$, as needed. □

Our procedure for computing the ciphertexts $\tilde{\mathbf{c}}_i$ mirrors Lemma 2. Specifically, we are given the ciphertext $\tilde{\mathbf{c}} = \tilde{\mathbf{c}}_0$ that encrypts $z = w_0 \bmod 2^{r+1}$, and we iteratively compute ciphertexts $\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2, \ldots$ such that $\tilde{\mathbf{c}}_i$ encrypts $w_i \bmod 2^{r-i+1}$. Eventually we get $\tilde{\mathbf{c}}_r$ that encrypts $w_r \bmod 2$, which is what we need (since the LSB of w_r is the r'th bit of z).

Note that most of the operations in Lemma 2 are carried out in $(\mathbb{Z}/2^{r+1}\mathbb{Z})$, and therefore can be evaluated homomorphically in our $(\mathbb{Z}/2^{r+1}\mathbb{Z})$-homomorphic cryptosystem. The only exception is the division by 2^i in Equation (1), and we now show how this division can also be evaluated homomorphically. To implement division we begin with an arbitrary ciphertext vector $\tilde{\mathbf{c}}$ that encrypts a plaintext element $a \in (\mathbb{Z}/2^j\mathbb{Z})[X]/F(X)$ (for some j) with respect to the level-0 key $\tilde{\mathbf{s}}$ and modulus q_0. Namely, we have the equality over $\mathbb{Z}[X]$:

$$(\langle \tilde{\mathbf{c}}, \tilde{\mathbf{s}}\rangle \bmod F(X)) = a + 2^j \cdot S + q_0 \cdot T$$

for some polynomials $S, T \in \mathbb{Z}[X]/F(X)$, where the norm of $a + 2^j S$ is much smaller than q_0. Assuming that a is divisible by 2 over the integers (i.e., all its coefficients are even) consider what happens when we multiply $\tilde{\mathbf{c}}$ by the integer

$(q_0+1)/2$ (which is the inverse of 2 modulo q_0). Then we have

$$\begin{aligned}
(\left\langle \tfrac{q_0+1}{2}\cdot\tilde{\mathbf{c}},\tilde{\mathbf{s}}\right\rangle \bmod F(X)) &= \tfrac{q_0+1}{2}\cdot(\langle\tilde{\mathbf{c}},\tilde{\mathbf{s}}\rangle \bmod F(X)) \\
&= \frac{(q_0+1)\cdot a}{2}+\frac{(q_0+1)\cdot 2^j\cdot S}{2}+\frac{q_0\cdot(q_0+1)\cdot T}{2} \\
&= (q_0+1)\cdot(a/2)\ +\ (q_0+1)\cdot 2^{j-1}S\ +\ q_0\cdot\tfrac{q_0+1}{2}\cdot T \\
&= a/2\ +\ 2^{j-1}\cdot S\ +\ q_0\cdot\left(a/2+2^{j-1}S+\tfrac{q_0+1}{2}T\right)
\end{aligned}$$

Clearly the coefficients of $a/2+2^{j-1}S$ are half the size of those of $a+2^jS$, hence they are much smaller than q_0. It follows that $\tilde{\mathbf{c}}'=[\tilde{\mathbf{c}}\cdot(q_0+1)/2]_{q_0}$ is a valid ciphertext that encrypts the plaintext $a/2\in(\mathbb{Z}/2^{j-1}\mathbb{Z})[X]/F(X)$ with respect to secret key $\tilde{\mathbf{s}}$ and modulus q_0.

The same argument shows that if a is divisible by 2^i over the integers (for some $i<j$) then $[\tilde{\mathbf{c}}\cdot((q_0+1)/2)^i]_{q_0}$ is a valid ciphertext encrypting $a/2^i\in(\mathbb{Z}/2^{j-i}\mathbb{Z})[X]/F(X)$. Combining this division-by-two procedure with homomorphic exponentiation mod 2^{r+1}, the resulting homomorphic bit-extraction procedure is described in Figure 1.

Bit-Extraction$(\tilde{\mathbf{c}},r,q_0)$:
Input: A ciphertext $\tilde{\mathbf{c}}$ encrypting a constant $b\in(\mathbb{Z}/2^{r+1}\mathbb{Z})$ w.r.t. secret key $\tilde{\mathbf{s}}$ and modulus q_0.
Output: A ciphertext $\tilde{\mathbf{c}}'$ encrypting $b\langle 0\rangle\oplus b\langle r-1\rangle\oplus b\langle r\rangle\in\mathbb{F}_2$ w.r.t. secret key $\tilde{\mathbf{s}}$ and modulus q_0.

1. Set $\tilde{\mathbf{c}}_0\leftarrow\tilde{\mathbf{c}}$ // $\tilde{\mathbf{c}}$ encrypt z w.r.t. $\tilde{\mathbf{s}}$
2. For $i=1$ to r
3. Set $\mathbf{acc}\leftarrow\tilde{\mathbf{c}}$ // $\mathbf{acc}$ is an accumulator
4. For $j=0$ to $i-1$ // Compute $z-\sum_j 2^j w_j^{i-1}$
5. Set $\mathbf{tmp}\leftarrow\mathsf{HomExp}(\tilde{\mathbf{c}}_j,2^{i-j})$ // Homomorphic exponentiation to the power 2^{i-j}
6. Set $\mathbf{acc}\leftarrow\mathbf{acc}-2^j\cdot\mathbf{tmp} \bmod q_0$
7. Set $\tilde{\mathbf{c}}_i\leftarrow\mathbf{acc}\cdot((q_0+1)/2)^i \bmod q_0$ // $\tilde{\mathbf{c}}_i$ encrypts $z\langle i\rangle$
8. Output $\tilde{\mathbf{c}}_0+\tilde{\mathbf{c}}_{r-1}+\tilde{\mathbf{c}}_r \bmod q_0$

$\mathsf{HomExp}(\tilde{\mathbf{c}},n)$ *uses native homomorphic multiplication to multiply* $\tilde{\mathbf{c}}$ *by itself* n *times. To aid exposition, this code assumes that the modulus and secret key remain fixed, else modulus-switching and key-switching should be added (and the level increased correspondingly to some* $i>0$*).*

Fig. 1. A Homomorphic Bit-Extraction Procedure.

3.2 Packing the Coefficients

Now that we have encryption of all the coefficients of a, we just need to "pack" all these coefficients back in one polynomial. Namely, we have encryption of the constant polynomials $a_0,a_1,\ldots$, and we want to get an encryption of the

polynomial $a(X) = \sum_i a_i X^i$. Since a is just a linear combination of the a_i's (with the coefficient of each a_i being the "scalar" $X^i \in (Z/2Z)[X]/\Phi_m$), we can just use the additive homomorphism of the cryptosystem to compute an encryption of a from the encryptions of the a_i's.

3.3 Lower-Degree Bit Extraction

As described in Figure 1, extracting the r'th bit requires computing polynomials of degree upto 2^r, here we describe a simple trick to lower this degree. Recall our simplified decryption process: we set $Z \leftarrow [\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_{2^{r+1}}$, and then recover $a = [Z\langle r \rangle + Z\langle r-1 \rangle + Z\langle 0 \rangle]_2$.

Consider what happens if we add q_L to all the odd coefficients in $\mathbf{c}$, call the resulting vector $\mathbf{c}'$: On one hand, now all the coefficients of $\mathbf{c}'$ are even. On the other hand, the coefficients of $Z' = \langle \mathbf{c}', \mathbf{s} \rangle \bmod \Phi_m(X)$ are still small enough to use Lemma 1 (since they are at most $c_m \cdot q \cdot \|\mathbf{s}\|_1$ larger than those of Z itself, where c_m is the ring constant of mod-$\Phi_m(X)$ arithmetic and $\|s\|_1$ is the l_1-norm of $\mathbf{s}$). Since $\mathbf{c}' = \mathbf{c} \pmod{q_L}$ then we have

$$[[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_{q_L}]_2 = [[\langle \mathbf{c}', \mathbf{s} \rangle \bmod \Phi_m(X)]_{q_L}]_2 = Z'\langle r \rangle + Z'\langle r \rangle - 1 + Z'\langle 0 \rangle$$

However, since $\mathbf{c}'$ is even then so is Z'. This means that $Z'\langle 0 \rangle = 0$, and if we divide Z' by two (over the integers), $Z'' = Z'/2$, then we have $[[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_{q_L}]_2 = Z''\langle r-1 \rangle \oplus Z''\langle r-2 \rangle$. We thus have a variation of the simple decryption formula that only needs to extract the $r-1$'st and $r-2$'nd bits, so it can be realized using polynomials of degree upto 2^{r-1}. Note that we can implement this variant of the decryption formula homomorphically, because Z' is even so an q_0-encryption of Z' can be easily converted into an encryption of $Z'/2$ (by multiplying by $\frac{q_0+1}{2}$ modulo q_0 as described in Section 3.1).

This technique can be pushed a little further, adding to $\mathbf{c}$ multiples of q so that it is divisible by 4, 8, 16, etc., and reducing the required degree correspondingly to 2^{r-2}, 2^{r-3}, 2^{r-4}, etc. The limiting factor is that we must maintain that $\langle \mathbf{c}', \mathbf{s} \rangle$ has coefficients sufficiently smaller than q_L^2, in order to be able to use Lemma 1. Clearly, if $\mathbf{c}' = \mathbf{c} + q\kappa$ where all the coefficients of κ are smaller than some bound B (in absolute value), then the coefficients of $\langle \mathbf{c}', \mathbf{s} \rangle$ can be larger than the coefficients of $Z = \langle \mathbf{c}, \mathbf{s} \rangle$ (in absolute value) by at most $c_m \cdot q \cdot B \cdot \|\mathbf{s}\|_1$. (Heuristically we expect the difference to depend on the l_2 norm of $\mathbf{s}$ more than its l_1 norm.)

If we choose our parameters such that the l_1-norm of $\mathbf{s}$ is below m, and work over a ring with $c_m = O(1)$, then the coefficients of Z can be made as small as $c_m \cdot m \cdot q$, and we can make the coefficients of κ as large as $B \approx q/(4c_m \cdot m)$ in absolute value while maintaining the invariant that the

coefficients of Z' are smaller than $q^2/4$ (which is what we need to be able to use Lemma 1). By choosing an appropriate κ, we can ensure that the least significant $\lfloor \log(q/(4c_m m)) \rfloor = r - \lceil \log(4c_m m) \rceil$ bits of $\mathbf{c}'$ are all zero. This means that we can implement bit extraction using only polynomials of degree at most $2^{\lceil \log(4c_m m) \rceil} < 8c_m m = O(m)$. (Heuristically, we should even be able to get polynomials of degree $O(\sqrt{m})$ since the l_2 norm of $\mathbf{s}$ is only $O(\sqrt{m})$.) Moreover if we assume that ring-LWE is hard even with a very sparse secret, then we can use a secret key with even smaller norm and get the same reduction in the degree of the bit-extraction routine.

4 Homomorphic Decryption with Packed Ciphertexts

The homomorphic decryption procedure from Section 3 is rather inefficient, mostly because we need to repeat the bit-extraction procedure from Figure 1 for each coefficient separately. Instead, we would like to pack many coefficients in one ciphertext and extract the top bits of all of them together. To this end we employ a batching technique, similar to [1, 11, 14], using Chinese remaindering over the ring of polynomials to pack many "plaintext slots" inside a single plaintext polynomial.

Recall that the BGV scheme is defined over a polynomial ring $R = \mathbb{Z}[X]/F(X)$. If the polynomial $F(X)$ factors modulo two into distinct irreducible polynomials $F_0(X) \times \cdots \times F_{\ell-1}(X)$, then, by the Chinese Remainder Theorem, the plaintext space factors into a product of finite fields $R_2 \cong \mathbb{F}_2[X]/F_0(X) \times \cdots \times \mathbb{F}_2[X]/F_{\ell-1}(X)$.

This factorization is used in [14, 1, 11] to "pack" a vector of ℓ elements (one from each $\mathbb{F}_2[X]/F_i(X)$) into one plaintext polynomial, which is then encrypted in one ciphertext; each of the ℓ components called a plaintext slot. The homomorphic operations (add/mult) are then applied to the different slots in a SIMD fashion. When $F(X)$ is the m-th cyclotomic polynomial, $F(X) = \Phi_m(X)$, then the field $\mathbb{Q}[X]/F(X)$ is Galois (indeed Abelian) and so the polynomials $F_i(X)$ all have the same degree (which we will denote by d). It was shown in [11] how to evaluate homomorphically the application of the Galois group on the slots, and in particular this enables homomorphically performing arbitrary permutations on the vector of slots in time quasi-linear in m. This, in turn, is used in [11] to evaluate arbitrary arithmetic circuits (of average width $\tilde{\Omega}(\lambda)$) with overhead only polylog(λ).

However, the prior work only mentions the case of plaintext spaces taken modulo a prime (in our case two), i.e. R_2. In this work we will need to also consider plaintext spaces which are given by a power of a prime, i.e. $R_{2^{r+1}}$ for some positive integer r. (We stress that by $R_{2^{r+1}}$ we really do mean $(\mathbb{Z}/2^t\mathbb{Z})[X]/F(X)$

and not $\mathbb{F}_{2^{r+1}}[X]/F(X)$.) In the full version [10] we show how the techniques from [11] extends also to this case. The "high brow" way of seeing this is to consider the message space modulo 2^{r+1} as the precision $r+1$ approximation to the 2-adic integers; namely we need to consider the localization of the field $K = \mathbb{Q}[X]/F(X)$ at the prime 2.

4.1 Using SIMD Techniques for Bootstrapping

Using the techniques from [11] for bootstrapping is not quite straightforward, however. The main difficulty is that the input and output of are not presented in a packed form: The input is a single q_L-ciphertext that encrypts a single plaintext polynomial a (which may or may not have many plaintext elements packed in its slots), and similarly the output needs to be a single ciphertext that encrypts the same polynomial a, but with respect to a larger modulus. (We stress that this is not an artifact of our "simpler decryption formula", we would need to overcome the same difficulty also if we tried to use these "SIMD techniques" to speed up bootstrapping under the standard approach of emulating the binary mod-q_L circuit.) Our "packed bootstrapping" procedure consists of the following steps:

1. Using the encryption of the q_L-secret-key with respect to the modulus q_0, we convert the initial q_L-ciphertext into a q_0-ciphertext encrypting the polynomial $Z \in (\mathbb{Z}/2^{r+1}\mathbb{Z})[X]/\Phi_m(X)$.
2. Next we apply a homomorphic inverse-DFT transformation to get encryption of polynomials that have the coefficients of Z in their plaintext slots.
3. Now that we have the coefficients of Z in the plaintext slots, we apply the bit extraction procedure to all these slots in parallel. The result is encryption of polynomials that have the coefficients of a in their plaintext slots.
4. Finally, we apply a homomorphic DFT transformation to get back a ciphertext that encrypts the polynomial a itself.

Below we describe each of these steps in more detail. We note that the main challenge is to get an efficient implementation of Steps 2 and 4.

4.2 Encrypting the q_L-Secret-Key

As in Section 3, we use an encryption scheme with underlying plaintext space modulo 2^{r+1} to encrypt the q_L-secret-key $\mathbf{s}$ under the q_0-secret-key $\tilde{\mathbf{s}}$. The q_L-secret-key is a vector $\mathbf{s} = (1, \mathfrak{s})$, where $\mathfrak{s} \in \mathbb{Z}[X]/\Phi_m(X)$ is an integer polynomial with small coefficients. Viewing these small coefficients as elements in $\mathbb{Z}/2^{r+1}\mathbb{Z}$, we encrypt $\mathfrak{s}$ as a q_0-ciphertext $\tilde{\mathbf{c}} = (\tilde{\mathfrak{c}}_0, \tilde{\mathfrak{c}}_1)$ with respect to the q_0-secret-key $\tilde{\mathbf{s}} = (1, \tilde{\mathfrak{s}})$, namely we have

$$[\langle \tilde{\mathbf{c}}, \tilde{\mathbf{s}} \rangle \bmod \Phi_m]_{q_0} = [\tilde{\mathfrak{c}}_0 + \tilde{\mathfrak{c}}_1 \cdot \tilde{\mathfrak{s}} \bmod \Phi_m]_{q_0} = 2^{r+1}\tilde{k} + \mathbf{s} \quad \text{(equality over } \mathbb{Z}[X])$$

for some polynomial $\tilde{k}$ with small coefficients.

4.3 Step One: Computing Z Homomorphically

Given a q_L-ciphertext $\mathbf{c} = (c_0, c_1)$ we recall from the public key the q_0 ciphertext $\tilde{\mathbf{c}} = (\tilde{\mathfrak{c}}_0, \tilde{\mathfrak{c}}_1)$ that encrypts $\mathfrak{s}$, then compute the mod-2^{r+1} inner product homomorphically by setting

$$\tilde{\mathbf{z}} = \big([c_0 + c_1\tilde{\mathfrak{c}}_0 \bmod \Phi_m]_{q_0},\ \ [c_1\tilde{\mathfrak{c}}_1 \bmod \Phi_m]_{q_0} \big). \tag{2}$$

We claim that $\tilde{\mathbf{z}}$ is a q_0-ciphertext encrypting our Z with respect to the secret key $\tilde{\mathbf{s}}$ (and plaintext space modulo 2^{r+1}). To see that, recall that we have the following two equalities over $\mathbb{Z}[X]$,

$$(c_0 + c_1\mathfrak{s} \bmod \Phi_m) = 2^{r+1}k + Z \ \text{ and } \ (\tilde{\mathfrak{c}}_0 + \tilde{\mathfrak{c}}_1\tilde{\mathfrak{s}} \bmod \Phi_m) = q_0\tilde{k} + 2^{r+1}\tilde{k}' + \mathfrak{s},$$

where $k, \tilde{k}, \tilde{k}' \in \mathbb{Z}[X]/\Phi_m$, the coefficients of $2^{r+1}k + Z$ are smaller than $2q_L^2 \ll q_0$, and the coefficients of $2^{r+1}\tilde{k}' + \mathfrak{s}$ are also much smaller than q_0. It follows that:

$$\begin{aligned}
(\langle \tilde{\mathbf{z}}, \tilde{\mathbf{s}} \rangle \bmod \Phi_m) \ &= \ [c_0' + c_1\tilde{\mathfrak{c}}_0 \bmod \Phi_m]_{q_0} + (\tilde{\mathfrak{s}} \cdot [c_1\tilde{\mathfrak{c}}_1 \bmod \Phi_m]_{q_0} \bmod \Phi_m) \\
&= \ (c_0' + c_1(\tilde{\mathfrak{c}}_0 + \tilde{\mathfrak{c}}_1\tilde{\mathfrak{s}}) \bmod \Phi_m) + q_0\kappa \\
&= \ (c_0' + c_1(2^{r+1}\tilde{k}' + \mathfrak{s}) \bmod \Phi_m) + q_0\kappa' \\
&= \ (c_0' + c_1\mathfrak{s} \bmod \Phi_m) + q_0\kappa' + 2^{r+1}(c_1 \cdot \tilde{k}' \bmod \Phi_m) \\
&= \ q_0\kappa' + 2^{r+1}(k + c_1\tilde{k}' \bmod \Phi_m) + Z \qquad \text{(equality over } \mathbb{Z}[X])
\end{aligned}$$

for some $\kappa, \kappa' \in \mathbb{Z}[X]/\Phi_m$. Moreover, since the coefficients of c_1 are smaller than $q_L \ll q_0$ then the coefficients of $2^{r+1}(k + c_1\tilde{k}' \bmod \Phi_m) + Z$ are still much smaller than q_0. Hence $\tilde{\mathbf{z}}$ is decrypted under $\tilde{\mathbf{s}}$ and q_0 to Z, with plaintext space 2^{r+1}.

4.4 Step Two: Switching to CRT Representation

Now that we have an encryption of the polynomial Z, we want to perform the homomorphic bit-extraction procedure from Figure 1. However, this procedure should be applied to each coefficient of Z separately, which is not directly supported by the native homomorphism of our cryptosystem. (For example, homomorphically squaring the ciphertext yields an encryption of the polynomial $Z^2 \bmod \Phi_m$ rather than squaring each coefficient of Z separately.) We therefore need to convert $\tilde{\mathbf{z}}$ to CRT-based "packed" ciphertexts that hold the coefficients of Z in their plaintext slots.

The system parameter m was chosen so that $m = \tilde{\Theta}(\lambda)$ and $\Phi_m(X)$ factors modulo 2 (and therefore also modulo 2^{r+1}) as a product of degree-d polynomials with $d = O(\log m)$, $\Phi_m(X) = \prod_{j=0}^{\ell-1} F_j(X) \pmod{2^{r+1}}$. This allows us to view the plaintext polynomial $Z(X)$ as having ℓ slots, with the j'th slot holding the value $Z(X) \bmod (F_j(X), 2^{r+1})$. This way, adding/multipliying/squaring the plaintext polynomials has the effect of applying the same operation on each of the slots separately.

In our case, we have $\phi(m)$ coefficients of $Z(X)$ that we want to put in the plaintext slots, and each ciphertext has only $\ell = \phi(m)/d$ slots, so we need d ciphertexts to holds them all. The transformation from the single ciphertext $\tilde{\mathbf{z}}$ that encrypts Z itself to the collection of d ciphertexts that hold the coefficients of Z in their slots is described in Section 4.7 below. (We describe that step last, since it is the most complicated and it builds on machinery that we develop for Step Four in Section 4.6.)

4.5 Step Three: Extracting the Relevant Bits

Once we have the coefficients of Z in the plaintext slots, we can just repeat the procedure from Figure 1. The input to the the bit-extraction procedure is a collection of some d ciphertexts, each of them holding $\ell = \phi(m)/d$ of the coefficients of Z in its ℓ plaintext slots. (Recall that we chose $m = \tilde{O}(\lambda)$ such that $d = O(\log m)$.) Applying the procedure from Figure 1 to these ciphertexts will implicitly apply the bit extraction of Lemma 2 to each plaintext slot, thus leaving us with a collection of d ciphertexts, each holding ℓ of the coefficients of a in its plaintext slots.

4.6 Step Four: Switching Back to Coefficient Representation

To finally complete the recryption process, we need to convert the d ciphertexts holding the coefficients of a in their plaintext slots into a single ciphertext that encrypts the polynomial a itself. For this transformation, we appeal to the result of Gentry et al. [11], which says that every depth-L circuit of average-width $\tilde{\Omega}(\lambda)$ and size T can be evaluated homomorphically in time $O(T) \cdot \text{poly}(L, \log \lambda)$, provided that the inputs and outputs are presented in a packed form. Below we show that the transformation we seek can be computed on cleartext by a circuit of size $T = \tilde{O}(m)$ and depth $L = \text{polylog}(m)$, and hence (since $m = \tilde{\Theta}(\lambda)$) it can be evaluated homomorphically in time $\tilde{O}(m) = \tilde{O}(\lambda)$.

To use the result of Gentry et al. we must first reconcile an apparent "type mismatch": that result requires that both input and output be presented in a packed CRT form, whereas we have input in CRT form but output in coefficient

form. We therefore must interpret the output as "something in CRT representation" before we can use the result from [11]. The solution is obvious: since we want the output to be a in coefficient representation, then it is a polynomial that holds the value $A_j = a \bmod F_j$ in the j'th slot for all j.

Hence the transformation that we wish to compute takes as input the coefficients of the polynomials $a(X)$, and produces as output the polynomials $A_j = a \bmod F_j$ for $j = 0, 1, \ldots, \ell - 1$. It is important to note that our output consists of ℓ values, each of them a degree-d binary polynomial. Since this output is produced by an arithmetic circuit, then we need a circuit that operates on degree-d binary polynomials, in other words an arithmetic circuit over $\mathbb{GF}(2^d)$. This circuit has $\ell \cdot d$ inputs (all of which happen to be elements of the base field $\mathbb{F}_2$), and ℓ outputs that belong to the extension field $\mathbb{GF}(2^d)$.

Theorem 1. *Fix $m \in \mathbb{Z}$, let $d \in \mathbb{Z}$ be the smallest such that $m|2^d - 1$, denote $\ell = \phi(m)/d$ and let $G \in \mathbb{F}_2[X]$ be a degree-d irreducible polynomial over $\mathbb{F}_2$ (that fixes a particular representation of $\mathbb{GF}(2^d)$). Let $F_0(X), F_1(X), \ldots, F_{\ell-1}(X)$ be the irreducible (degree-d) factors of the m-th cyclotomic polynomial $\Phi_m(X)$ modulo 2.*

Then there is an arithmetic circuit Π_m over $\mathbb{F}_2[X]/G(X) = \mathbb{GF}(2^d)$ with $\phi(m)$ inputs $a_0, a_1, \ldots, a_{\phi(m)-1}$ and ℓ outputs $z_0, z_1, \ldots, z_{\ell-1}$, for which the following conditions hold:

- *When the inputs are from the base field ($a_i \in \mathbb{F}_2 \; \forall i$) and we denote $a(X) = \sum_i a_i X^i \in \mathbb{F}_2[X]$, then the outputs satisfy $z_j = a(X) \bmod (F_j(X), 2) \in \mathbb{F}_2[X]/G(X)$.*
- *Π_m has depth $O(\log m)$ and size $O(m \log m)$.*

The proof is in the full version. An immediate corollary of Theorem 1 and the Gentry et al. result [11, Thm. 3], we have:

Corollary 2. *There is an efficient procedure that given d ciphertexts, encrypting d polynomials that hold the coefficients of a in their slots, computes a single ciphertext encrypting a. The procedure works in time $O(m) \cdot \text{polylog}(m)$ (and uses at most $\text{polylog}(m)$ levels of homomorphic evaluation).*

4.7 Details of Step Two

The transformation of Step Two is roughly the inverse of the transformation that we described above for Step Four, with some added complications. In this step, we have the polynomial $Z(X)$ over the ring $\mathbb{Z}/2^{r+1}\mathbb{Z}$, and we view it as defining ℓ plaintext slots with the j'th slot containing $B_j \stackrel{\text{def}}{=} Z \bmod (F_j, 2^{r+1})$. Note that the B_j's are degree-d polynomials, and we consider them as elements

in the "extension ring" $R^d_{2^{r+1}} \stackrel{\text{def}}{=} \mathbb{Z}[X]/(G(X), 2^{r+1})$ (where G is some fixed irreducible degree-d polynomial modulo 2^{r+1}).

Analogous to Theorem 1, we would like to argue that there is an arithmetic circuit over $R^d_{2^{r+1}}$ that get as input the B_j's (as elements of $R^d_{2^{r+1}}$), and outputs all the coefficients of Z (which are elements of the base ring $\mathbb{Z}/2^{r+1}\mathbb{Z}$). Then we could apply again to the result of Gentry et al. [11] to conclude that this circuit can be evaluated homomorphically with only polylog overhead.

For the current step, however, the arithmetic circuit would contain not only addition and multiplication gates, but also Frobenius map gates. Namely, gates $\rho_k(\cdot)$ (for $k \in \{1, 2, \ldots, d-1\}$) computing the functions

$$\rho_k\big(u(X)\big) \;=\; u(X^{2^k}) \bmod (G(X), 2^{r+1}).$$

It was shown in [11] that arithmetic circuits with Frobenius map gates can also be evaluated homomorphically with only polylog overhead. The Frobenius operations being simply an additional automorphism operation which can be applied homomorphically to ciphertexts.

Theorem 2. *Fix $m, r \in \mathbb{Z}$, let $d \in \mathbb{Z}$ be the smallest such that $m|2^d - 1$, denote $\ell = \phi(m)/d$ and let $G(X)$ be a degree-d irreducible polynomial over $\mathbb{Z}/2^{r+1}\mathbb{Z}$ (that fixes a particular representation of $R^d_{2^{r+1}}$). Let $F_0(X), F_1(X), \ldots, F_{\ell-1}(X)$ be the irreducible (degree-d) factors of the m-th cyclotomic polynomial $\Phi_m(X)$ modulo 2^{r+1}.*

Then there is an arithmetic circuit $\Psi_{m,r}$ with Frobenius-map gates over $R^d_{2^{r+1}}$ that has ℓ input $B_0, B_1, \ldots, B_{\ell-1}$ and $\phi(m)$ outputs $Z_0, Z_1, \ldots, Z_{\phi(m)-1}$, for which the following conditions hold:

- *On any inputs $B_0, \ldots, B_{\ell-1} \in R^d_{2^{r+1}}$, the outputs of $\Psi_{m,r}$ are all in the base ring, $Z_i \in \mathbb{Z}/2^{r+1}\mathbb{Z}$ $\forall i$. Moreover, denoting $Z(X) = \sum_i Z_i X^i$, it holds that $Z(X) \bmod (F_j(X), 2^{r+1}) = B_j$ for all j.*
- *Π_m has depth $O(\log m + d)$ and size $O(m(d + \log m))$.*

The proof is in the full version. As before, a corollary of Theorem 2 and the result from [11], is the following:

Corollary 3. *There is an efficient procedure that given a single ciphertext encrypting Z' outputs d ciphertexts encrypting d polynomials that hold the coefficients of Z' in their plaintext slots. The procedure works in time $\tilde{O}(m)$ (and uses at most polylog(m) levels of homomorphic evaluation).*

4.8 An Alternative Variant

The procedure from Section 4.7 works in time $\tilde{O}(m)$, but it is still quite expensive. One alternative is to put in the public key not just one ciphertext encrypting the q_L-secret-key $\mathfrak{s}$, but rather d ciphertexts encrypting polynomials that hold the coefficients of $\mathfrak{s}$ in their plaintext slots. Then, rather than using the simple formula from Equation (2) above, we evaluate homomorphically the inner product of $\mathbf{s} = (1, \mathfrak{s})$ and $\mathbf{c} = (c_0, c_1)$ modulo $\Phi_m(X)$ and 2^{r+1}. This procedure will be even faster if instead of the coefficients of $\mathfrak{s}$ we encrypt their transformed image under length-m DFT. Then we can compute the DFT of c_1 (in the clear), multiply it homomorphically by the encrypted transformed $\mathfrak{s}$ (in SIMD fashion) and then homomorphically compute the inverse-DFT and the reduction modulo Φ_m. Unfortunately this procedure still requires that we compute the reduction mod-$\Phi_m(X)$ homomorphically, which is likely to be the most complicated part of bootstrapping. Finding a method that does not require this homomorphic polynomial modular reduction is an interesting open problem.

Acknowledgments. The first and second authors are sponsored by DARPA and ONR under agreement number N00014-11C-0390. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

The third author is sponsored by DARPA and AFRL under agreement number FA8750-11-2-0079. The same disclaimers as above apply. He is also supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant COED–EP/I03126X, and by a Royal Society Wolfson Merit Award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the European Commission or EPSRC.

References

1. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science (ITCS'12)*, 2012. Available at `http://eprint.iacr.org/2011/277`.
2. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS'11*. IEEE Computer Society, 2011.

3. Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.
4. John William Scott Cassels. *Local Fields*, volume 3 of *LMS Student Texts*. Cambridge University Press, 1986.
5. Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in Cryptology - EUROCRYPT'10*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010. Full version available on-line from `http://eprint.iacr.org/2009/616`.
6. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.
7. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.
8. Craig Gentry and Shai Halevi. Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. In *FOCS'11*. IEEE Computer Society, 2011.
9. Craig Gentry and Shai Halevi. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011.
10. Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping for fully homomorphic encryption. `http://eprint.iacr.org/2011/680`, 2011.
11. Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012. Full version at `http://eprint.iacr.org/2011/566`.
12. Ron Rivest, Leonard Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–180, 1978.
13. Nigel P. Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography - PKC'10*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010.
14. Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. Manuscript at `http://eprint.iacr.org/2011/133`, 2011.

Ring Switching in BGV-Style Homomorphic Encryption
(Preliminary Version)

Craig Gentry Shai Halevi Nigel P. Smart

January 29, 2015

Abstract

BGV-style homomorphic encryption schemes over polynomial rings, rely for their security on rings of very large dimension. This large dimension is needed because of the large modulus-to-noise ratio in the key-switching matrices that are used for the top few levels of the evaluated circuit. However, larger noise (and hence smaller modulus-to-noise ratio) is used in lower levels of the circuit, so from a security standpoint it is permissible to switch to lower-dimension rings, thus speeding up the homomorphic operations for the lower levels of the circuit. However, implementing such ring-switching is nontrivial, since these schemes rely on the ring algebraic structure for their homomorphic properties.

A basic ring-switching operation was used by Brakerski, Gentry and Vaikuntanathan, over polynomial rings of the form $\mathbb{Z}[X]/(X^{2^n}+1)$, in the context of bootstrapping. In this work we generalize and extend this technique to work over any cyclotomic ring and show how it can be used not only for bootstrapping but also during the computation itself (in conjunction with the "packed ciphertext" techniques of Gentry, Halevi and Smart.)

Note: A later version of this work, with a substantially different transformation, appears in SCN 2012.

Acknowledgments

The first and second authors are supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center (DoI/NBC) contract number D11PC20202. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

The third author is supported by the National Science Foundation under CAREER Award CCF-1054495, by the Alfred P. Sloan Foundation, and by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under Contract No. FA8750-11-C-0098. The views expressed are those of the authors and do not necessarily reflect the official policy or position of the National Science Foundation, the Sloan Foundation, DARPA or the U.S. Government.

The fourth author is supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant COED–EP/I03126X, and by a Royal Society Wolfson Merit Award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the European Commission or EPSRC.

Contents

1 Introduction

The last year has seen a rapid advance in the state of fully homomorphic encryption; yet despite these advances the existing schemes are still too inefficient for most practical purposes. In this paper we make another step forward in making such schemes more efficient. In particular we present a technique to reduces the dimension of the ring needed for homomorphic computation of the lower levels of a circuit. Our techniques apply to homomorphic encryption schemes over polynomial rings, such as the scheme of Brakerski et al. [6, 7, 5], as well as the variants due to Lòpez-Alt et al. [15] and Brakerski [4].

The most efficient variants of all these schemes work over polynomial rings of the form $\mathbb{Z}[X]/F(X)$, and in all of them the ring dimension (which is the degree of $F(X)$) must be set high enough to ensure security: To be able to handle depth-L circuits, these schemes must use key-switching matrices with modulus-to-noise ratio of $2^{\tilde{\Omega}(L\cdot\mathrm{polylog}(\lambda))}$, hence the ring dimension must also be $\tilde{\Omega}(L\cdot\mathrm{polylog}(\lambda))$ (even if we assume that ring-LWE is hard to within fully exponential factors).[1] In practice, the ring dimension for moderately deep circuits can easily be many thousands. For example, to be able to evaluate AES homomorphically, Gentry et al. used in [14] circuits of depth $L \geq 50$, with corresponding ring-dimension of over 50000.

As homomorphic operations proceed, the noise in the ciphertext grows (or the modulus shrinks, if we use the modulus-switching technique from [7, 5]), hence reducing the modulus-to-noise ratio. Consequently, it becomes permissible to start using lower-dimension rings in order to speed up further homomorphic computation. However, in the middle of the computation we already have evaluated ciphertexts over the big ring, and so we need a method for transforming these into small-ring ciphertexts that encrypt the same thing. Such a "ring switching" procedure was described by Brakerski et al. [5], in the context of reducing the ciphertext-size prior to bootstrapping. The procedure in [5], however, is specific to polynomial rings of the form $R_{2^n} = \mathbb{Z}[X]/(X^{2^{n-1}}+1)$, and moreover by itself it cannot be combined with the "packed evaluation" techniques of Gentry et al. [12]. Extending this procedure is the focus of this work.

1.1 Our Contribution

In this work we present two complementary techniques:

- We extend the procedure from [5] to any cyclotomic ring $R_m = \mathbb{Z}[X]/\Phi_m(X)$ for a composite m. This is important, since the tools from [12] for working with "packed" ciphertexts require that we work with an odd parameter m. For $m = u \cdot w$, we show how to break a ciphertext over the big ring R_m into a collection of $u = m/w$ ciphertexts over the smaller ring $R_w = \mathbb{Z}[X]/\Phi_w(X)$, such that the plaintext-polynomial encrypted in the original big-ring ciphertext can be recovered as a simple linear function of the plaintext-polynomials encrypted in the smaller-ring ciphertexts.

- We then show how to take a "packed" big-ring ciphertext that contains many plaintext elements in its plaintext slots, and distribute these plaintext elements among the plaintext slots of several small-ring ciphertexts. If the original big-ring ciphertext was "sparse" (i.e., if only few of its plaintext slots were used), then our technique yields just a small number of small-ring ciphertexts, only as many as needed to fit all the used plaintext slots.

The first technique on its own may be useful in the context of bootstrapping, but it is not enough to achieve our goal of reducing the computational overhead by switching to small-ring ciphertexts, since we

[1] The schemes from [5, 4] can replace large rings by using higher-dimension vectors over smaller rings. But their most efficient variants use big rings and low-dimension vectors, since the complexity of their key-switching step is quadratic in the dimension of these vectors.

still need to show how to perform homomorphic operations on the resulting small-ring ciphertexts. This is achieved by utilizing the second technique. To demonstrate the usefulness of the second technique, consider the application of homomorphic AES computation [14], where the original big-ring ciphertext contains only 16 plaintext elements (corresponding to the 16 bytes of the AES state). If the small-ring ciphertexts has 16 or more plaintext slots, then we can convert the original big-ring ciphertext into a single small-ring ciphertext containing the same 16 bytes in its slots, then continue the computation on this smaller ciphertext.

1.2 An Overview of the Construction

Our starting point is the polynomial composition technique of Brakerski et al. [5]. When $m = u \cdot w$ then a polynomial of degree up to $m-1$, $a(X) = \sum_{i=0}^{m-1} a_i X^i$, can be broken into u polynomials of degree up to $w-1$ by splitting the coefficients of a according to their index modulo u. Namely, denoting by $a_{(k)}$ the polynomial with coefficients $a_k, a_{k+u}, a_{k+2u}, \ldots$, we have

$$a(X) \;=\; \sum_{k=0}^{u-1}\sum_{j=0}^{w-1} a_{k+uj}X^{k+uj} \;=\; \sum_{k=0}^{u-1} X^k \sum_{j=0}^{w-1} a_{k+uj}X^{uj} \;=\; \sum_{k=0}^{u-1} X^k a_{(k)}(X^u). \tag{1}$$

We note that this "very syntactic" transformation (of splitting the coefficients of a big-ring polynomial into several small-ring polynomials) has the following crucial algebraic properties:

1. The end result is a collection of "parts" $a_{(k)}$, all from the small ring R_w (which is a sub-ring of the big ring R_m, since $w|m$).
2. Recalling that $f(x) \mapsto f(x^u)$ is an embedding of R_w inside R_m, we have the property that the original a can be recovered as a simple linear combination of (the embedding of) the parts $a_{(k)}$.
3. Moreover the transformation $T(a) = \langle a_{(0)}, \ldots, a_{(u-1)} \rangle$ is linear, and as such it commutes with the linear operations inside the decryption formula of BGV-type schemes: If $\mathfrak{s}$ is a big-ring secret key and c is (part of) a big-ring ciphertext, then decryption over the big ring includes computing $a = \mathfrak{s} \cdot c \in R_m$ (and later reducing a mod q and mod 2). Due to linearity, the parts of a can be expressed in terms of the tensor product between the parts of $\mathfrak{s}$ and c. Namely, $T(\mathfrak{s} \cdot c)$ is some linear function (over the small ring R_w) of $T(\mathfrak{s}) \otimes T(c)$.

In addition to these algebraic properties, in the case considered in [5] where m, w are powers of two, it turns out that this transformation also possess the following geometric property:

4. If a is a low-norm element in R_m, then all the parts $a_{(k)}$ in $T(a)$ are low-norm elements in R_w.

The importance of this last property stems from the fact that a valid ciphertext in a BGV-type homomorphic encryption scheme must have a low noise, namely its inner-product with the unknown secret key must be a low-norm polynomial. Property 3 above is used to convert a big-ring ciphertext encrypting a (relative to a big-ring secret key $\mathfrak{s}$) into a collection of "syntactically correct" small-ring ciphertexts encrypting the $a_{(k)}$'s (relative to the small-ring secret key $T(\mathfrak{s})$), and Property 4 is used to argue that these small-ring ciphertexts are indeed valid.

When attempting to apply the same transformation for m, w that are not powers of two, it turns out that the algebraic properties must all still hold, but the geometric property may not. One plausible solution is to find a different transformation $T(\cdot)$ for breaking a big-ring element into a vector of small-ring elements, that has all the properties 1-4 above, even when m, w are not powers of two. In the current work, however,

we stick to the same transformation $T(\cdot)$ as in [5], and address the problem with the geometric property by "lifting" everything from the big ring $R_m = \mathbb{Z}[X]/\Phi_m(X)$ to the even bigger ring $C_m = \mathbb{Z}[X]/(X^m - 1)$, using techniques similar to [12, 9].

The reason that lifting to C_m helps, is that over the bigger ring C_m, the linear combination from Equation (1) is in fact a "direct sum", in the sense that every coefficient of the left-hand side comes from exactly one of the terms on the right. Thus if the result is a low-norm polynomial then all the summands must also be low-norm polynomials, which is what we need.[2]

A Key-Switching Optimization. One source of inefficiency in the ring-switching procedure of Brakerski et al. [5] is that using the tensor product $T(\mathfrak{s}) \otimes T(c)$ amounts essentially to having u small-ring ciphertexts, each of which is a dimension-u vector over the small ring. Brakerski et al. point out that we can use key-switching/dimension-reduction to convert these high dimension ciphertexts into low-dimension ciphertexts over the small ring, but processing u ciphertexts of dimension u requires work quadratic in u. Instead, here we describe an alternative procedure that saves a factor of u in running time:

Before using $T(\cdot)$ to break the ciphertext into pieces, we apply key-switching over the big ring to get a ciphertext with respect to another secret key that happens to belong to the small ring R_w (which we note again is a sub-ring of R_m). The transformation $T(\cdot)$ has the additional property that when applied to a small-ring element $\mathfrak{s}' \in R_w \subset R_m$, the resulting vector $T(\mathfrak{s}')$ over R_w has just a single non-zero element (namely $\mathfrak{s}'$ itself). Hence $\mathsf{T}(\mathfrak{s}') \otimes T(c)$ is the same as just $\mathfrak{s}' \cdot T(c)$, and this lets us work directly with low-dimension ciphertexts over the small ring (as opposed to ciphertexts of dimension u). This is described in Section 3.1, where we prove that key-switching into a key from the small subring is secure as long as ring-LWE [16] is hard in that small subring.

Packed Ciphertexts. As sketched so far, the ring-switching procedure lets us convert a big-ring ciphertext encrypting a polynomial $a \in R_m$ into a collection of u' small-ring ciphertexts encrypting the parts $a_{(k)} \in R_w$. However, coming in the middle of homomorphic evaluation, we may need to get small-ring ciphertexts encrypting things other than the $a_{(k)}$'s. Specifically, if the original polynomial a encodes several plaintext elements in its plaintext slots (as in [19, 12]), we may want to get encryption of small-ring polynomials that encode the same elements in their slots.

We note that the plaintext elements encoded in the polynomial $a \in R_m$ are the evaluations $a(\zeta_i)$ where the ζ_i's are primitive m-th roots of unity in some extension field $\mathbb{F}_{2^d}$. (Equivalently, the evaluations $a(\zeta_i)$ can also be described as $a \bmod \mathfrak{p}_i$, where $\mathfrak{p}_i$ is some prime ideal in the ring R_m — specifically the ideal generated by $\{2, X - \zeta_i\}$. Noting that these prime ideals are exactly the factors of 2 in R_m, this evaluation representation over $GF(2^d)$ is nothing more than Chinese-Remaindering over the prime factors of 2 in R_m.)

Similarly, the plaintext elements encoded in a polynomial $b \in R_w$ are the evaluations $b(\tau_j)$ with the τ_j's are primitive w-th roots of unity (equivalently the residues of b relative to the prime ideal factors of 2 in R_w). Our goal, then, is to decompose a big-ring ciphertext encrypting a into small-ring ciphertexts encrypting some b_t's, such that for every i there are some t, j for which $b_t(\tau_j) = a(\zeta_i)$.

To that end, we interpret Equation (1) as expressing the value of a at an arbitrary point X as a linear combination of the values of the $a_{(k)}$'s at the point X^u (with coefficients $1, X, X^2, \ldots, X^{u-1}$). Observing that if ζ in an m-th root of unity then $\tau = \zeta^u$ is a w-th root of unity, we thus obtain a method of expressing the values of a in the m-th roots of unity as linear combinations of the values of the $a_{(k)}$'s in the w-th roots of unity. In Lemma 6 in Section 4 we show how to express, under some conditions on m and w, the

[2] In the power-of-two setting considered in [5], the same "direct sum" argument can be applied directly in the big ring R_{2^n}, hence they do not need the "lifting" technique.

coefficients of the linear combination from Equation (1) as (low norm) polynomials in the τ_j's. This allows us to compute the encryption of the b_t's that we seek as low-weight linear combination of the encryption of the $a_{(k)}$'s that we obtained before.

A bird-eye view of this last transformation is that the linear transformation $T(a)$ that we used to break the plaintext big-ring element into a vector of small-ring parts has the side-effect of inducing some linear transformation (over $\mathbb{F}_{2^d}$) on the contents of the plaintext slots. Hence after we apply T, we compute homomorphically the inverse linear transformation, thereby recovering the original content.

2 Notation and Preliminaries

Below we define the various algebraic structures that we need for this work. In this paper we will be utilizing various rings at different points, all will be associated to rings of roots of unity. Below let m, q be arbitrary positive integers. Let $\Phi_m(X)$ denote the m'th cyclotomic polynomial (i.e., $\Phi_m(X) = \prod_{i\in(\mathbb{Z}/mZ)^*}(X - \zeta_m^i)$, where ζ_m is the complex primitive m'th root of unity, $\zeta_m = e^{2\pi i/m}$). Recalling that Φ_m is an integer polynomial, we define the following rings:

$$\begin{array}{lcllcl} R_m & = & \mathbb{Z}[X]/\Phi_m(X), & C_m & = & \mathbb{Z}[X]/(X^m-1) \\ R_{m,q} & = & \mathbb{Z}[X]/(\Phi_m(X),q), & C_{m,q} & = & \mathbb{Z}[X]/(X^m-1,q) \end{array}$$

We will be interested in cyclotomic rings for a composite $m = u \cdot w$.

The size of polynomials. Throughout this work we frequently refer to "low norm polynomials". The norm that we use to measure the size of polynomials is the l_2 norm of their coefficient vectors, i.e. for a polynomial f we set $\mathsf{norm}(f) = \sqrt{\sum f_i^2}$. (Most of our treatment is not very sensitive to the choice of the particular norm function.) We informally say that a polynomial in $R_{m,q}$ or $C_{m,q}$ has low norm when its norm is much smaller than the parameter q.

The ring constant c_m. We sometime need to switch back and forth between $R_{m,q}$ and $C_{m,q}$ while maintaining "low norm" polynomials. For every integer m there exists a constant c_m that bounds the increase in norm due to reduction modulo $\Phi_m(X)$. Namely, for every polynomial f of degree up to $m-1$ is holds that $\mathsf{norm}(f \bmod \Phi_m) \le c_m \cdot \mathsf{norm}(f)$.

Empirically, the constants c_m for the parameters m that we work with is rather small (ranging between 2 and 50 for typical values). But in principle for very smooth m's the constant c_m can be super-polynomial in m. For the rest of the paper we always assume that our parameters are chosen so that $q \gg c_m$, so that we can take "low norm" polynomials in C_m and reduce them modulo Φ_m without increasing the norm too much (relative to q). Note that ring constant c_m is different, but related to, the associated ring constant from [8, 12].

2.1 RLWE-based BGV Cryptosystems

Below and throughout this work we denote by $[z]_q$ the reduction of the integer z modulo the positive integer q into the symmetric interval $(-q/2, q/2)$. In our initial ring-LWE-based BGV cryptosystem, secret keys and ciphertexts are 2-vectors over $R_{m,q}$ for some odd system parameter q, and moreover the secret key has the form $\mathbf{s} = (1, \mathfrak{s})$ where $\mathfrak{s} \in R_m$ is a low-norm polynomial (e.g., with coefficients in $\{-1, 0, 1\}$). The native plaintext space for our initial BGV scheme will be $R_{m,2}$, namely binary polynomials modulo $\Phi_m(X)$.

A valid ciphertext $\mathbf{c} = (c_0, c_1) \in (R_{m,q})^2$ that encrypts the plaintext polynomial $a \in R_{m,2}$ with respect to $\mathbf{s} = (1, \mathfrak{s})$ satisfies the equality (over R_m)

$$[\langle \mathbf{c}, \mathbf{s} \rangle]_q = [c_0 + \mathfrak{s} \cdot c_1]_q = a + 2e, \tag{2}$$

for some low-norm polynomial $e \in R_m$. Note that by $[c_0 + \mathfrak{s} \cdot c_1]_q$ we mean reducing each of the coefficients of the polynomial $c_0 + \mathfrak{s} \cdot c_1 \in R_m$ into the interval $(-q/2, q/2)$. Decryption is then just computing $[c_0 + \mathfrak{s} \cdot c_1]_q$, then reducing modulo 2 to recover the plaintext polynomial a.

Throughout the paper we will switch back and forth between different rings. We will maintain the invariant that valid ciphertexts always satisfy Equation (2), but the ring over which this equation is evaluated (specifically the meaning of $\mathfrak{s} \cdot c_1$) will vary. In the input to the ring-switching procedure we will have a ciphertext where that equality holds over R_m, at the end we will have the output ciphertexts for which the equality holds over R_w, and in various intermediate points we will have that equality holding over C_m or C_w.

2.2 Plaintext Arithmetic

Following [19, 5, 12, 13, 14] we consider plaintext polynomials $a \in R_{m,2}$ as encoding vectors of plaintext elements from some finite field $\mathbb{F}_{2^d}$, where d is the order of 2 in the group $(\mathbb{Z}/m\mathbb{Z})^*$. (This implies that d divides $\phi(m)$, and also that $\mathbb{F}_{2^d}$ contains primitive m-th roots of unity.) Denoting $\ell = \phi(m)/d$, we can identify polynomials in $R_{m,2}$ with ℓ-vectors of elements from $\mathbb{F}_{2^d}$. The specific mapping between polynomials and vectors that we use is as follows:

Consider the quotient group $(\mathbb{Z}/m\mathbb{Z})^*/\langle 2 \rangle$ (which has exactly ℓ elements), and fix a specific set of representatives for this quotient group, $T_m = \{t_1, t_2, \ldots, t_\ell\} \subseteq (\mathbb{Z}/m\mathbb{Z})^*$, containing exactly one element from every conjugacy class in $(\mathbb{Z}/m\mathbb{Z})^*/\langle 2 \rangle$.[3] Also fix a specific primitive m-th root of unity $\zeta \in \mathbb{F}_{2^d}$, and we identify each polynomial $a \in R_{m,2}$ with the ℓ-vector consisting of $a(\zeta^t)$ for all $t \in T_m$:

$$a \in R_{m,2} \longleftrightarrow \left\langle a(\zeta^{t_1}), \ldots, a(\zeta^{t_\ell}) \right\rangle \in (\mathbb{F}_{2^d})^\ell.$$

Showing that this is indeed a one-to-one mapping is a standard exercise. In one direction clearly from a we can compute all the values $a(\zeta^{t_i})$. In the other direction we use the fact that since the coefficients of a are all in the base field $\mathbb{F}_2$ then $a(X^2) = a(X)^2$ for any $X \in \mathbb{F}_{2^d}$. In particular from $a(\zeta^{t_i})$ we can compute $a(\zeta^{2t_i})$, $a(\zeta^{4t_i})$, $a(\zeta^{8t_i})$, and so on. Since T_m is a complete set of representatives for the quotient group $(\mathbb{Z}/m\mathbb{Z})^*/\langle 2 \rangle$, then we can get this way the evaluations of $a(\zeta^j)$ for all the indexes $j \in (\mathbb{Z}/m\mathbb{Z})^*$. This gives us the evaluation of a in $\phi(m)$ different points, from which we can interpolate a itself.

We thus view the evaluation of the plaintext polynomial in ζ^{t_j} as the j'th "plaintext slot", and note that arithmetic operations in the ring $R_{m,2}$ act on the plaintext slots in a SIMD manner, namely point-wise adding or multiplying the elements in the slots.

We can equivalently view this mapping as Chinese remaindernig represntation (which makes the one-to-one argument and the SIMD property obvious, but requires careful choises for the represenation of $\mathbb{F}_{2^d}$ in the different plaintext slots).

2.3 Breaking Polynomials in Parts

As sketched in the introduction, our approach is rooted at the technique for assembling a high-degree polynomial from low-degree parts by interleaving the coefficients of the parts. Alternatively, we can view this

[3] In other words, the sets $T_m, 2T_m, 4T_m, \ldots 2^{d-1}T_m$ are all disjoint, and their union is the entire group $(\mathbb{Z}/m\mathbb{Z})^*$.

as breaking a high-degree polynomial into small-degree parts. Recall that for a polynomial a of degree up to $m-1$, and for any integer $u < m$, we can break a into u parts of degree less than $w = \lceil m/u \rceil$, denoted $a_{(0)}, \ldots, a_{(u-1)}$, by splitting the coefficients of a according to their index mod u, thus obtaining

$$a(X) \;=\; \sum_{k=0}^{u-1}\sum_{j=0}^{w-1} a_{k+uj}\cdot X^{k+uj} \;=\; \sum_{k=0}^{u-1} X^k \cdot \Big(\sum_{j=0}^{w-1} a_{k+uj}\cdot X^{uj}\Big) \;=\; \sum_{k=0}^{u-1} X^k \cdot a_{(k)}(X^u).$$

Of particular interest to us will be the case where $m = u \cdot w$, where working with w-degree polynomials that are evaluated at X^u allows us to move between big rings and small rings. The following lemma will be useful later in the paper.

Lemma 1. *Let m, w be positive integers such that w divides m, and let $u = m/w$. Also let $\Phi_m(X)$, $\Phi_w(X)$ be the m-th and w-th cyclotomic polynomials, respectively.*

a. Consider three polynomials $f(X), g(X), h(X)$ of degree at most $\phi(w)-1$. If $h(X) \equiv f(X)\cdot g(X) \pmod{\Phi_w(X)}$ then $h(X^u) \equiv f(X^u)\cdot g(X^u) \pmod{\Phi_m(X)}$.

b. Consider three polynomials $f(X), g(X), h(X)$ of degree at most $w-1$. If $h(X) \equiv f(X)\cdot g(X) \pmod{X^w-1}$ then $h(X^u) \equiv f(X^u)\cdot g(X^u) \pmod{X^m-1}$.

Proof. **a.** Since $h(X) \equiv f(X)\cdot g(X) \pmod{\Phi_w(X)}$ then for every primitive w-th root of unity τ (say, over the complex field) we have $h(\tau) = f(\tau)\cdot g(\tau)$. Let us denote $\tilde{f}(X) = f(X^u) \bmod \Phi_m(X)$, $\tilde{g}(X) = g(X^u) \bmod \Phi_m(X)$, and $\tilde{h}(X) = h(X^u) \bmod \Phi_m(X)$, then for every primitive m-th root of unity ζ we have

$$\tilde{f}(\zeta)\cdot\tilde{g}(\zeta) \;=\; f(\zeta^u)\cdot g(\zeta^u) \;\stackrel{(\star)}{=}\; h(\zeta^u) \;=\; \tilde{h}(\zeta)$$

where the equality $(\star)$ follows since ζ^u is a primitive w-th of unity whenever ζ is a primitive m-th of unity. Since $\tilde{f}\cdot\tilde{g}$ has the same evaluations as $\tilde{h}$ on all the primitive m-th roots of unity then it follows that $\tilde{f}\cdot\tilde{g} \equiv \tilde{h} \pmod{\Phi_m}$, as needed.

b. The proof is identical to Part a, except that we consider all w-th and m-th roots of unity, not just the primitive roots. □

3 The Basic Ring-Switching Procedure

Given a big-ring ciphertext $\mathbf{c} \in (R_{m,q})^2$, encrypting a plaintext polynomial $a \in R_{m,2}$ relative to a big-ring secret key $\mathfrak{s} \in R_m$, our goal is roughly to come up with u small-ring ciphertexts $\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{u-1} \in (R_{w,q})^2$ with $\mathbf{c}_i$ encrypting the part $a_{(i)} \in R_{w,2}$, all relative to some small ring secret key $\mathfrak{s}' \in R_w$. The basic procedure consists of the following steps:

1. **Key-switch.** We use the BGV key-switching method from [5] to switch into a "low-dimension" secret key, still over the big ring $R_{m,q}$. The "low-dimension" key is $\mathfrak{s}'' \in R_m$, where $\mathfrak{s}''$ has nonzero coefficients only for powers X^i where $i \equiv 0 \pmod{u}$. That is, we have $\mathfrak{s}''_{(0)} = \mathfrak{s}'$ and $\mathfrak{s}''_{(i)} = 0$ for all $i > 0$ (in other words $\mathfrak{s}''(X) = \mathfrak{s}'(X^u)$).

2. **Lift.** Next we lift the resulting ciphertext from the big ring $R_{m,q}$ to the even bigger ring $C_{m,q}$, using the delayed-reduction technique of Gentry et al. [12]. As described in Section 3.2, the new ciphertext encrypts over the bigger ring $C_{m,q}$ a plaintext polynomial a' related to a, still relative to the big-ring secret key $\mathfrak{s}''$.

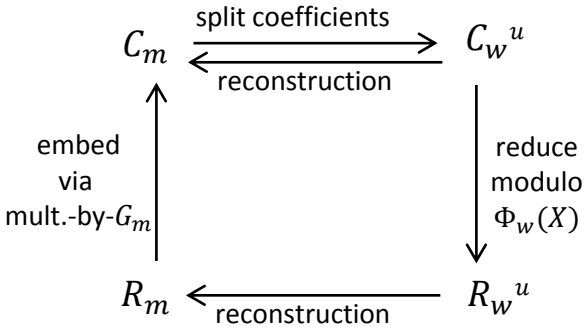

Figure 1: The transformation used to map elements between the different spaces.

3. **Break.** Now we can break the bigger-ring ciphertext into a collection of u intermediate-ring ciphertexts (i.e., pairs over $C_{w,q}$), such that the k'th ciphertext is a valid encryption of the k'th part of a' (i.e., $a'_{(k)} \in C_{w,2}$). All these ciphertexts are valid (over $C_{w,q}$) with respect to the small-ring secret key $\mathfrak{s}'$.

4. **Reduce.** Finally we reduce all the intermediate ring ciphertexts modulo $(\Phi_w(X), q)$, thereby getting small ring ciphertexts over $R_{w,q}$, valid relative to $\mathfrak{s}'$.

We observe that the small ring ciphertext that we get this way may not encrypt the parts $a_{(k)}$ of the original polynomial a. Rather, we will show that they encrypt some other polynomials $\tilde{a}_k$, which are defined as $\tilde{a}_k = a'_{(k)} \bmod (\Phi_w, 2)$. We will show, however, that these plaintext polynomials $\tilde{a}_k$ satisfy the same relation to the original plaintext polynomial, namely $a(X) \equiv \sum_k X^k \cdot \tilde{a}_k(X^u) \pmod{\Phi_m, 2}$, which is all we need for our application.

3.1 Switching to a Low-Dimension Key

To enable this transformation, we include in the public key a "key switching matrix", essentially encrypting the old key $\mathfrak{s}$ under the new low-dimension key $\mathfrak{s}''$. Note that using such a low-dimension secret key has security implications (since it severely reduces the dimension of the underlying LWE problem). In our case, however, the whole point of switching to a smaller ring is to get lower dimension, so we do not sacrifice anything new. Indeed, we show below that assuming the hardness of the decision-ring-LWE problem [16] over the ring $R_{w,q}$, the key-switching matrix in the public key is indistinguishable from a uniformly random matrix over $R_{m,q}$ (even for a distinguisher that knows the old secret key $\mathfrak{s}$).

The ring-LWE problem in $R_{w,q}$. We denote the secret-key and error-distributions prescribed in the ring-LWE problem in $R_{w,q}$ by $\mathcal{S}_w$ and $\mathcal{E}_w$, respectively. (E.g., these could be low-variance Gaussians in R_w rounded modulo q, or some distributions involving the dual as in [16].) We also denote the uniform distribution on $R_{w,q}$ by $\mathcal{U}_w$. For a fixed random secret $\mathfrak{s}' \leftarrow \mathcal{S}_w$, the ring-LWE problem in $R_{w,q}$ is given many pairs (γ_i, δ_i) with $\gamma_i \leftarrow \mathcal{U}_w$, to distinguish the cases where the δ_i's are chosen as $\delta_i = \mathfrak{s}' \cdot \gamma_i + \eta_i$ with η_i from the case where they are chosen uniformly at random $\delta_i \leftarrow \mathcal{U}_w$.

The key-switching matrix. Let $\mathfrak{s} \in R_m$ be the old big-ring secret key, and $\mathfrak{s}' \in R_w$ be the small-ring secret-key that we want to switch into (where $\mathfrak{s}'$ was chosen from the secret-key distribution $\mathcal{S}_w$). Define the new big-ring low-dimension key $\mathfrak{s}'' \in R_m$ as the unique polynomial of degree less than m such that $\mathfrak{s}''_{(0)} = \mathfrak{s}'$ and $\mathfrak{s}''_{(k)} = 0$ for all $k > 0$. In other words, $\mathfrak{s}''(X) = \tilde{\mathfrak{s}}(X^u)$, i.e., the coefficients $\mathfrak{s}''_0, \mathfrak{s}''_u, \mathfrak{s}''_{2u}, \ldots$ are exactly the coefficients of $\mathfrak{s}'$, and all the other coefficients of $\mathfrak{s}''$ are zero.

For our key-switching matrix we use the following distribution of "error vectors" in $R_{w,q}$: We first draw independently at random u low-norm polynomials from the ring-LWE error distribution, $\eta_{(k)} \leftarrow \mathcal{E}_w$, then assemble from the $\eta_{(k)}$'s a single error polynomial $\epsilon'(X) = \sum_{k=0}^{u-1} X^k \cdot \eta_{(k)}(X^u)$, and output $\epsilon = \epsilon' \bmod (\Phi_m, q)$. That is, we have the distribution

$$\mathcal{E}_m \;=\; \left\{ \eta_{(0)}, \ldots, \eta_{(u-1)} \leftarrow \mathcal{E}_w, \;\; \mathsf{output} \sum_{k=0}^{u-1} X^k \cdot \eta_{(k)}(X^u) \bmod (\Phi_m, q) \right\}$$

Note that ϵ' before the reduction mod (Φ_m, q) has degree smaller than $\phi(w) \cdot u < m$, and its norm-squared is the sum of norm-squared of the $\epsilon_{(k)}$'s. Hence ϵ' is a low-norm polynomial, and the norm of ϵ after the reduction is larger by at most a factor of c_m (c_m is the ring constant for R_m), so ϵ too is a low norm polynomial.[4]

Given the old key $\mathfrak{s} \in R_{m,q}$ and the new $\mathfrak{s}' \in R_{w,q}$, we draw at random $l = \lceil \log q \rceil$ elements from the error distribution $\epsilon_0, \ldots, \epsilon_{l-1} \leftarrow \mathcal{E}_m$, and the columns of our key-switching matrix are the pairs

$$\left\{ (\beta_i, \alpha_i)^t : \alpha_i \leftarrow \mathcal{U}_m, \;\; \beta_i = 2^i \mathfrak{s} - (\mathfrak{s}'' \cdot \alpha_i + 2\epsilon_i) \bmod (\Phi_m, q) \right\},$$

where $\mathcal{U}_m$ is the uniform distribution over the big ring $R_{m,q}$. (Note that even if the secret-key and error distributions over the small ring involce the "dual lattice" as in [16], the β's are still going to be in the big ring, because all their parts $\beta_{(k)}$ are in the small ring.)[5]

Since the errors ϵ_i have low-norm, this is a functional key-switching matrix, as described in [7]. Given an $\mathfrak{s}$-ciphertext $\mathbf{c} = (c_0, c_1)$ we decompose c_1 into its bit representation, thus getting an l-vector of polynomials with 0-1 coefficients. Multiplying that vector by the key-switching matrix and adding c_0 to the first coordinate we get a new ciphertext $\mathbf{c}' = (c'_0, c'_1)$ with respect to the new low-dimension big-ring key $\mathfrak{s}''$. As for security, we prove the following lemma.

Lemma 2. *If the decision ring-LWE problem over the ring $R_{w,q}$ is hard, then the key-switching matrix above is indistinguishable from a uniformly random $2 \times l$ matrix with all the entries drawn independently from $\mathcal{U}_m$. The indistinguishability holds even if the distinguisher gets as input the old secret key $\mathfrak{s} \in R_m$.*

Proof. Our goal is to show that under the hardness of ring-LWE in R_w, it is infeasible to distinguish the case where the β_i's where chosen as prescribed in the scheme from the case where they are uniformly random according to $\mathcal{U}_m$. That is, we show that an adversary $\mathcal{A}$ that given the old secret key $\mathfrak{s}$ and the matrix of (β_i, α_i)'s can distinguish between these two distributions, can be used to solve the ring-LWE problem in the small ring $R_{w,q}$.

The reduction. A ring-LWE distinguisher $\mathcal{B}$ gets $l \cdot u$ pairs $(\gamma_{i,k}, \delta_{i,k})$, for $i = 0, 1, \ldots, l-1$ and $k = 0, 1, \ldots, u-1$, where the $\gamma_{i,k}$'s are uniform in $R_{w,q}$ and the $\delta_{i,k}$'s are either set as $\mathfrak{s}' \cdot \gamma_{i,k} + \eta_{i,k}$, for

[4] This argument can be refined to eliminate the dependence on the "smallness" of c_m, see Remark 1 at the end of the section.

[5] We could alternatively use the key-switching variant from [14] where the "matrix" consists of a single column $(\beta, \alpha)^t$), but with respect to a largest modulus $Q \approx q^2 \cdot m$. The proof of security would then depend on the hardness of ring-LWE in $R_{w,Q}$ rather than in $R_{w,q}$.

$\eta \leftarrow \mathcal{E}_w$, or chosen at random $\delta_{i,k} \leftarrow \mathcal{U}_w$. $\mathcal{B}$ begins by choosing an "old secret key" in the big ring $\mathfrak{s} \in R_{m,q}$ (according to whatever distribution the scheme specifies). Then $\mathcal{B}$ assembles the α_i's and β_i's by setting

$$\alpha_i(X) = 2\sum_{k=0}^{u-1} X^k \cdot \gamma_{i,k}(X^u) \bmod (\Phi_m, q) \text{ and } \beta_i(X) = 2^i \cdot \mathfrak{s} - 2 \cdot \sum_{k=0}^{u-1} X^k \cdot \delta_{i,k}(X^u) \bmod (\Phi_m, q).$$

Finally, $\mathcal{B}$ runs the adversary $\mathcal{A}$ on $\mathfrak{s}$ and the matrix with columns $(\beta_i, \alpha_i)^t$ and outputs whatever $\mathcal{A}$ does.

Analysis. We observe that when we have polynomials $f_0, f_1, \ldots, f_{u-1} \in R_{w,q}$ and we set $g(X) = \sum_{k=0}^{u-1} X^k f_k(X^u) \bmod (\Phi_m, q)$, then the coefficients of g are related to those of the f_k's via a $(\phi(w) \cdot u) \times \phi(m)$ matrix of full rank (i.e., rank $\phi(m)$) over $\mathbb{Z}/q\mathbb{Z}$. When the f_k's are drawn from $\mathcal{U}_w$ then all their coefficients are uniform in $\mathbb{Z}/q\mathbb{Z}$, and therefore so are all the coefficients of g.

Applying this observation to the reduction above, since the $\gamma_{i,k}$'s are uniform in the small ring $R_{w,q}$ then the α_i's are set as twice a uniform element in the big ring $R_{m,q}$, which is also uniform since q is odd. Similarly, if the $\delta_{i,k}$'s are uniform in $R_{w,q}$ then also the β_i's are uniform in the big ring $R_{m,q}$. On the other hand, if the $\delta_{i,k}$'s are chosen as $\delta_{i,k} = \mathfrak{s}' \cdot \gamma_{i,k} + \eta_{i,k} \bmod (\Phi_w, q)$, with $\eta_{i,k} \leftarrow \mathcal{E}_w$, then we have

$$\begin{aligned}
\beta_i(X) &\equiv 2^i \cdot \mathfrak{s}(X) - 2\sum_{k=0}^{u-1} X^k \cdot \delta_{i,k}(X^u) \\
&= 2^i \cdot \mathfrak{s}(X) - 2 \cdot \sum_{k=0}^{u-1} X^k \cdot \overbrace{\left[(\mathfrak{s}' \cdot \gamma_{i,k} + \eta_{i,k}) \bmod (\Phi_w, q)\right](X^u)}^{\delta_{i,k} \text{ evaluated at } X^u} \\
&\overset{(\star)}{\equiv} 2^i \cdot \mathfrak{s}(X) - 2 \cdot \sum_{k=0}^{u-1} X^k \cdot \overbrace{\left[\mathfrak{s}' \cdot \gamma_{i,k} + \eta_{i,k}\right]}^{\text{no modular reduction}} (X^u) \\
&\equiv 2^i \cdot \mathfrak{s}(X) - \underbrace{\mathfrak{s}'(X^u)}_{\mathfrak{s}''(X)} \cdot \underbrace{2 \cdot \sum_{k=0}^{u-1} X^k \cdot \gamma_{i,k}(X^u)}_{\alpha_i(X)} - \underbrace{2\sum_{k=0}^{u-1} X^k \cdot \eta_{i,k}(X^u)}_{\epsilon_i(X)} \pmod{\Phi_m, q},
\end{aligned}$$

where the equality $(\star)$ follows from Lemma 3 (part a). In this case the α_i's are still uniformly random, but the ϵ_i's are drawn exactly from our error distribution $\mathcal{E}_m$ in the big ring $R_{m,q}$. This completes the proof. □

3.2 Lifting to the Bigger Ring $C_{m,q}$

To lift the ciphertexts from the big ring $R_{m,q}$ to the bigger ring $C_{m,q}$, we use the "delayed reduction" technique of Gentry et al. (from the full version of [12]), which builds on the following lemma:

Lemma 3. *([12, Lemma 12]) For any integer m there is an integer polynomial G_m of degree $\leq m-1$, such that $G_m(\alpha) = m$ for every complex primitive m-th root of unity α, and $G_m(\beta) = 0$ for every complex non-primitive m-th root of unity β. Moreover the Euclidean norm of G_m's coefficient vector is $\sqrt{m \cdot \phi(m)}$.*

Denoting $Q_m(X) = (X^m - 1)/\Phi_m(X)$, then $G_m(X) \equiv m \pmod{\Phi_m}$ and $G_m(X) \equiv 0 \pmod{Q_m}$. We can use polynomial Chinese remaindering to construct G_m from its remainders modulo $\Phi_m(X)$ and $Q_m(X)$. Since $G_m(X) \equiv 0 \pmod{Q_m}$ then we can use G_m to "lift" any equality modulo Φ_m to an equality modulo $X^m - 1$. Namely, if we have $f \equiv g \pmod{\Phi_m}$ then we also have $G \cdot f \equiv G \cdot g \pmod{X^m - 1}$. Specifically for the decryption formula, we start from a valid big-ring ciphertext that

satisfies the formula $c_0 + c_1 \cdot \mathfrak{s}'' \equiv a + 2e + q\kappa \pmod{\Phi_m}$ (for some low-norm polynomial e and a quotient polynomial κ), then multiply both sides by G_m to obtain

$$(G_m \cdot c_0) + (G_m \cdot c_1) \cdot \mathfrak{s}'' \equiv 2(G_m \cdot e) + (G_m \cdot a) + q(G_m \cdot \kappa) \pmod{X^m - 1}.$$

Assuming that $q \gg m$, the products $G_m \cdot e \bmod (X^m - 1)$ and $G_m \cdot a \bmod (X^m - 1)$ are both low-norm. Thus, denoting $c'_0 = G_m \cdot c_0 \bmod (X^m - 1)$ and $c'_1 = G_m \cdot c_1 \bmod (X^m - 1)$, we get that the ciphertext (c'_0, c'_1) is a valid encryption over the bigger ring C_m of $a' = G_m \cdot a \bmod (X^m - 1, 2)$, relative to the secret key $\mathfrak{s}''$. (We note that upon decryption, one can recover the original plaintext polynomial a, simply by reducing a' modulo $(\Phi_m(X), 2)$, this yields $[m \cdot a]_2 = a$, because $G_m(X) \equiv m \pmod{\Phi_m}$ and m is odd.)

3.3 Breaking The Ciphertext into Parts

After the transformation of the previous step, our ciphertext consists of a pair (c, d) of polynomials in the bigger ring $C_{m,q} = \mathbb{Z}[X]/((X^m - 1), q)$. This ciphertext is valid with respect to the low-dimension secret key $\mathfrak{s}''$ of degree smaller than $\phi(m)$, satisfying $\mathfrak{s}''_{(0)} = \mathfrak{s}' \in R_{w,q}$ and $\mathfrak{s}''_{(1)} = \mathfrak{s}''_{(2)} = \cdots = \mathfrak{s}''_{(u-1)} = 0$, in other words $\mathfrak{s}''(X) = \mathfrak{s}'(X^u)$. Breaking c, d into their parts $c_{(k)}, d_{(k)}$, we then have the following lemma.

Lemma 4. *The polynomials $c_{(k)}$ and $d_{(k)}$ are such that the following equality holds over $\mathbb{Z}[X]$:*

$$\big[c + d \cdot \mathfrak{s}'' \bmod (X^m - 1, q)\big](X) \;=\; \sum_{k=0}^{u-1} X^k \cdot \big[c_{(k)} + d_{(k)} \cdot \mathfrak{s}' \bmod (X^w - 1, q)\big](X^u).$$

(In the above equality, we have on both sides polynomials that are reduced to a lower degree and have their coefficients reduced modulo q, then evaluated at X or X^u.)

Proof. Recall that decryption over $C_{m,q}$ calls for computing $z = c + d \cdot \mathfrak{s}'' \bmod (X^m - 1)$, then reducing z modulo q and then modulo 2. Breaking the polynomials c, d and $\mathfrak{s}''$ into parts, we can write:

$$\begin{aligned}
(d \cdot \mathfrak{s}'')(X) &= \sum_{k=0}^{2u-2} \sum_{\substack{i,j \text{ s.t.} \\ i+j=k}} X^k \cdot d_{(i)}(X^u) \cdot \mathfrak{s}''_{(j)}(X^u) \\
&= \sum_{k=0}^{u-1} X^k \cdot \left(\sum_{\substack{i,j \text{ s.t.} \\ i+j=k}} d_{(i)}(X^u) \cdot \mathfrak{s}''_{(j)}(X^u) + \sum_{\substack{i,j \text{ s.t.} \\ i+j=k+u}} X^u \cdot d_{(i)}(X^u) \cdot \mathfrak{s}''_{(j)}(X^u) \right) \\
&\overset{(\star)}{=} \sum_{k=0}^{u-1} X^k \cdot d_{(k)}(X^u) \cdot \mathfrak{s}''_{(0)}(X^u) \;=\; \sum_{k=0}^{u-1} X^k \cdot d_{(k)}(X^u) \cdot \mathfrak{s}'(X^u)
\end{aligned}$$

where the equality $(\star)$ follows since $\mathfrak{s}''_{(j)} = 0$ for $j > 0$ and $d_{(i)} = 0$ for $i \geq u$. This implies also that

$$\begin{aligned}
(c + d \cdot \mathfrak{s}'')(X) &= \sum_{k=0}^{u-1} X^k \cdot c_{(k)}(X^u) + \sum_{k=0}^{u-1} X^k \cdot d_{(k)}(X^u) \cdot \mathfrak{s}'(X^u) \\
&= \sum_{k=0}^{u-1} X^k \cdot \big[c_{(k)} + d_{(k)} \cdot \mathfrak{s}'\big](X^u)
\end{aligned}$$

Recall from Lemma 3 (part b) that whenever we have $h(X) \equiv f(X) \cdot g(X) \pmod{X^w - 1}$ then also $h(X^u) \equiv f(X^u) \cdot g(X^u) \bmod (X^m - 1)$. Hence we have

$$(c + d \cdot \mathfrak{s}'')(X) \equiv \sum_{k=0}^{u-1} X^k \cdot \big[c_{(k)} + d_{(k)} \cdot \mathfrak{s}' \bmod (X^w - 1)\big](X^u) \pmod{X^m - 1},$$

and since the right-hand side of the last equality is a polynomial of degree less than m, then we get the following equality holding over $\mathbb{Z}[X]$:

$$\big[c + d \cdot \mathfrak{s}'' \bmod (X^m - 1, q)\big](X) \;=\; \sum_{k=0}^{u-1} X^k \cdot \big[c_{(k)} + d_{(k)} \cdot \mathfrak{s}' \bmod (X^w - 1, q)\big](X^u).$$

We note that in the above equality, we have on both sides polynomials that are reduced to a lower degree and have their coefficients reduced modulo q, then evaluated at X or X^u. However, once we perform these modular reduction on both sides, then both polynomials have degrees less than m and coefficients smaller than $q/2$ in absolute value, and since they are congruent modulo $((X^m - 1), q)$ then they must be identical. □

Size of Polynomials. Importantly, the sum on the right-hand side of the last equality is a "direct sum", in the sense that the k'th summand has non-zero coefficients only in powers X^i such that $i = k \pmod{u}$. This means that each coefficient in the sum comes from exactly one of the summands. This, in turn, implies that the norm-squared of the left-hand side is the sum of norm-squared of the terms on the right-hand side. Hence if the left-hand side has low norm, then also *every summand on the right* must have low norm.

We stress that this "direct sum" argument is the reason why we lift our ciphertext to the bigger ring $C_{m,q}$. This argument does not apply when working modulo Φ_m, thus without lifting we could not have used the fact that the left-hand side has low norm to argue that all the terms on the right have low norm.

Ciphertexts in the intermediate ring $C_{w,q}$. Consider now the u intermediate-ring ciphertexts over $C_{w,q}$:

$$\mathbf{c}_0 = (c_{(0)}, d_{(0)}), \;\; \mathbf{c}_1 = (c_{(1)}, d_{(1)}), \;\; \ldots, \;\; \mathbf{c}_{u-1} = (c_{(u-1)}, d_{(u-1)}).$$

Since the bigger-ring ciphertext (c, d) was a valid encryption of $a' = G_m \cdot a \bmod (X^m - 1, 2)$ over $C_{m,q}$ with respect to secret key $\mathfrak{s}''$, we know that we have $[c + d \cdot \mathfrak{s}'' \bmod (X^m - 1, q)] = 2e' + a'$ for some low-norm error e'. Let us denote $b' = 2e' + a'$. From the equalities above (and the "direct sum" argument), we know that the k'th part of b', namely $b'_{(k)} = 2e'_{(k)} + a'_{(k)}$, is obtained as $b'_{(k)} = [c_{(k)} + d_{(k)} \cdot \mathfrak{s}' \bmod (X^w - 1, q)]$. As $e'_{(k)}$ is a low-norm error term, we conclude that the vectors $\mathbf{c}_k$ are valid encryption of the parts $a'_{(k)}$ over $C_{w,q}$ with respect to secret key $\mathfrak{s}'$. Thus we have shown that valid ciphertexts encrypting the parts $a'_{(k)}$ of a' (over the intermediate ring $C_{w,q}$ with respect to $\mathfrak{s}'$) can be obtained simply by breaking the polynomials c, d into their parts.

3.4 Reducing to the Small Ring $R_{w,q}$

Now that we have valid ciphertext $(c_{(k)}, d_{(k)})$ encrypting the parts $a'_{(k)}$ over the intermediate ring $C_{w,q}$ relative to $\mathfrak{s}'$, it only remains to reduce them into the small ring $R_{w,q}$. We do this simply by reducing each of the element $(c_{(k)}, d_{(k)})$ modulo (Φ_w, q), i.e. we set $\tilde{c}_k = c_{(k)} \bmod (\Phi_w, q)$ and $\tilde{d}_k = d_{(k)} \bmod (\Phi_w, q)$.

Lemma 5. *The ciphertext $(\tilde{c}_k, \tilde{d}_k)$ is an encryption (over $R_{w,q}$) of the plaintext $\tilde{a}_k = a'_{(k)} \bmod (\Phi_w, 2) \in R_{w,2}$.*

Proof. Recall that for all k we have the equality (over $\mathbb{Z}[X]$)

$$c_{(k)} + d_{(k)} \cdot \mathfrak{s}' \bmod (X^w - 1, q) = 2e'_{(k)} + a'_{(k)}$$

for a low-norm error term $e'_{(k)}$. Denoting $b'_{(k)} = 2e'_{(k)} + a'_{(k)}$, we have that $b'_{(k)}$ is a low-norm polynomial in $C_{w,q}$.

Let us now denote $\tilde{b}_k = (b'_{(k)} \bmod \Phi_w)$ (without reduction modulo q). Since the $b'_{(k)}$'s are low-norm then so are the $\tilde{b}_k$ (because reduction modulo Φ_w increases the norm by at most a factor of the ring constant c_w). This means that $\tilde{b}_k$ has norm much smaller than q, so it is already reduced modulo q. In other words, we also have $\tilde{b}_k = b'_{(k)} \bmod (\Phi_w, q)$.

Observe that $\tilde{a}_k = (a'_{(k)} \bmod \Phi_w) + 2 \cdot \mu_k$ for some low-norm μ_k's. The μ_k's have low norm because $\tilde{a}_k$ has low norm (being a 0-1 polynomial) and also $(a'_{(k)} \bmod \Phi_w)$ has low norm (being at most c_w time more than the norm of the 0-1 polynomial $a'_{(k)}$). Next we argue that for all k we have $\tilde{b}_k = 2\tilde{e}_k + \tilde{a}_k$ for a low-norm error terms $\tilde{e}_k \in R_{w,q}$. This follows because

$$\begin{aligned}\tilde{b}_k = (b'_{(k)} \bmod \Phi_w) \;&=\; (2 \cdot e'_{(k)} + a'_{(k)} \bmod \Phi_w) \;=\; (2 \cdot e'_{(k)} \bmod \Phi_w) + (a'_{(k)} \bmod \Phi_w) \\ &= 2 \cdot (e'_{(k)} \bmod \Phi_w) + \tilde{a}_k - 2 \cdot \mu_k \;=\; 2 \cdot \underbrace{((e'_{(k)} \bmod \Phi_w) - \mu_k)}_{\tilde{e}_k} + \tilde{a}_k,\end{aligned}$$

Finally, we obtain:

$$\begin{aligned}(\tilde{c}_k + \tilde{d}_k \cdot \mathfrak{s}' \bmod (\Phi_w, q)) &= (c_{(k)} + d_{(k)} \cdot \mathfrak{s}' \bmod (\Phi_w, q)) \\ &= (b'_{(k)} \bmod (\Phi_w, q)) \;=\; \tilde{b}_k \;=\; 2 \cdot \tilde{e}_k + \tilde{a}_k\end{aligned}$$

In other words, since $\tilde{e}_k$ has low norm then the pair $(\tilde{c}_k, \tilde{d}_k)$ is a valid ciphertext over $R_{w,q}$ with respect to secret key $\mathfrak{s}'$, encrypting the plaintext polynomial $\tilde{a}_k \in R_{w,2}$. □

What are the $\tilde{a}_k$'s? At this point we are done converting the original big-ring ciphertext encrypting $a \in R_{m,2}$ into a collection of valid small-ring ciphertexts encrypting the $\tilde{a}_k$'s. But how are these $\tilde{a}_k$'s related to the original plaintext polynomial a? Ideally we would have liked the $\tilde{a}_k$ to be the parts of a (i.e. $\tilde{a}_k = a_{(k)}$), but this is not necessarily what we get. Still, we show that we can recover the original polynomial a from the $\tilde{a}_k$'s via the same assembly formula,

$$a(X) \;=\; \sum_{k=0}^{u-1} X^k \cdot \tilde{a}_k(X^u) \bmod (\Phi_m, 2).$$

To show that we first observe that on both sides of the equation are 0-1 polynomials of degree less than $\phi(m)$, so to demonstrate equality it is enough to show that they agree when evaluated at $\phi(m)$ different points (from any field of our choice). In particular, we now show that they agree on all the primitive m'th roots of unity over the finite field $\mathbb{F}_{2^d}$. For this we recall the following basic facts:

1. The field $\mathbb{F}_{2^d}$ contains primitive m'th roots of unity, and if $\zeta \in \mathbb{F}_{2^d}$ is a primitive m'th roots of unity then ζ^u is a primitive w'th root of unity.

2. Since $G_m \equiv m \equiv 1 \pmod{\Phi_m, 2}$, then $[G_m \bmod 2](\zeta) = 1$ for every primitive m'th root of unity $\zeta \in \mathbb{F}_{2^d}$. Since $a' = G_m \cdot a \bmod (X^m - 1, 2)$, it then follows that $a'(\zeta) = a(\zeta)$ for every primitive m'th root of unity $\zeta \in \mathbb{F}_{2^d}$.

3. Since $\tilde{a}_k = a'_{(k)} \bmod (\Phi_w, 2)$, then $\tilde{a}_k(\tau) = a'_{(k)}(\tau)$ for every primitive w'th root of unity $\tau \in \mathbb{F}_{2^d}$.

Putting all of these facts together, and using the assembly formula for a' from the parts $a'_{(k)}$, we get for every primitive m'th root of unity $\zeta \in \mathbb{F}_{2^d}$:

$$a(\zeta) \overset{\text{Fact 2}}{=} a'(\zeta) = \sum_{k=0}^{u-1} \zeta^k \cdot a'_{(k)}(\zeta^u) \overset{\text{Facts 1,3}}{=} \sum_{k=0}^{u-1} \zeta^k \cdot \tilde{a}_k(\zeta^u)$$

Remark 1. *If we use the delayed reduction technique from [12, Appendix E] then we can keep everything relative to $X^m - 1$ and $X^w - 1$ and we do not need to rely on the smallness of the ring constants c_m, c_w. The key-switching matrices will remain modulo Φ_m, however.*

4 Homomorphic Computation in the Small Ring

So far we have shown how to break a big-ring ciphertext, encrypting some big-ring polynomial $a \in R_{m,2}$, into a collection of u small-ring ciphertexts encrypting small-ring polynomials $\tilde{a}_0, \tilde{a}_1, \ldots, \tilde{a}_{u-1} \in R_{w,2}$, that are "related" to the original plaintext polynomial a. Namely a can be constructed as a particular big-ring linear combination of the $\tilde{a}_k$'s, $a(X) = \sum_k X^k \cdot \tilde{a}_k(X^u) \bmod (\Phi_m, 2)$.

This, however, still falls short of our goal of speeding-up homomorphic computation by switching to small-ring ciphertexts. Indeed we have not shown how to use the encryption of the $\tilde{a}_k$'s for further homomorphic computation. Following the narrative of SIMD homomorphic computation from [19, 12, 13, 14], we view the big-ring plaintext polynomial a as an encoding in the big ring of several plaintext elements from the extension field $\mathbb{F}_{2^d}$ (with d the order of 2 in $(\mathbb{Z}/m\mathbb{Z})^*$). We therefore wish to obtain small-ring ciphertexts encrypting small-ring polynomials that encode of the same underlying $\mathbb{F}_{2^d}$ elements.

One potential "algebraic issue" with this goal, is that it may not always be possible to embed $\mathbb{F}_{2^d}$ elements inside small-ring polynomials from $R_{w,2}$. Recall that the extension degree d is determined by the order of 2 in $(\mathbb{Z}/m\mathbb{Z})^*$. But the order of 2 in $(\mathbb{Z}/w\mathbb{Z})^*$ may be smaller than d, in general it will be some d' that divides d. If $d' < d$ then we can only embed elements of the sub-field $\mathbb{F}_{2^{d'}}$ in small-ring polynomials from $R_{w,2}$, and not the $\mathbb{F}_{2^d}$ elements that we have encoded in the big-ring polynomial a. For most of this section we only consider the special case where the order of 2 in both $(\mathbb{Z}/m\mathbb{Z})^*$ and $(\mathbb{Z}/w\mathbb{Z})^*$ is the same d. We discuss possible extensions to the general case at the end of the section.

Even for the special case where the order of 2 in $(\mathbb{Z}/m\mathbb{Z})^*$ and $(\mathbb{Z}/w\mathbb{Z})^*$ is the same (and hence the "plaintext slots" in the small ring contain elements from the same extension field as those in the big ring), we still need to tackle the issue that big ring polynomials have more plaintext slots than small ring polynomials. Specifically, big-ring polynomials have $\ell_m = \phi(m)/d$ slots, whereas small-ring polynomials only have $\ell_w = \phi(w)/d$ slots. The solution here is simple: we just partition the slots in the original big-ring polynomial a into $\ell_m/\ell_w = \phi(m)/\phi(w)$ groups, each consisting of ℓ_w slots. For each group we then construct a small-ring ciphertext, encrypting a small-ring polynomial that encodes the plaintext slots from that group.

One advantage of this approach is that if the original plaintext polynomial a was "sparsely populated", holding only a few plaintext elements in its slots, then we can reduce the number of small ring ciphertexts that we generate to the bear minimum number needed to hold these few plaintext slots. A good example for this scenario is the computation of the AES circuit in [14]: Since there are only 16 bytes in the AES

state, we only use 16 slots in the plaintext polynomial a. In this case, as long as we have at least 16 slots in small-ring polynomials, we can continue working with a single small-ring ciphertext (as opposed to the u ciphertexts that the technique of the previous section gives us).

4.1 Ring-Switching with Plaintext Encoding

Below we describe our method for converting the plaintext encoding between the different rings, for the special case where the order of 2 is the same in $(\mathbb{Z}/m\mathbb{Z})^*$ and $(\mathbb{Z}/w\mathbb{Z})^*$. As explained in Section 2.2, each plaintext slot in the big-ring polynomial is associated with a conjugacy class of 2 in $(\mathbb{Z}/m\mathbb{Z})^*$ (equivalently, an element in the quotient group $\mathcal{Q}_m = (\mathbb{Z}/m\mathbb{Z})^*/\langle 2\rangle$), and similar association holds between plaintext slots in small-ring polynomials and elements of the quotient group $\mathcal{Q}_w = (\mathbb{Z}/w\mathbb{Z})^*/\langle 2\rangle$. We thus begin by relating the structures and representations of these two quotient groups. Below let $T_w = \{t'_1, \ldots, t'_{\ell_w}\} \subseteq (\mathbb{Z}/w\mathbb{Z})^*$ be a representative set for $\mathcal{Q}_w$. i.e., a set containing exactly one element from each conjugacy class in $(\mathbb{Z}/w\mathbb{Z})^*$, ordered arbitrarily.

Clearly, since w divides m then $(\mathbb{Z}/m\mathbb{Z})^*$ consists of $\phi(m)/\phi(w)$ copies of $(\mathbb{Z}/w\mathbb{Z})^*$. That is, $(\mathbb{Z}/m\mathbb{Z})^*$ can be partitioned into $\phi(m)/\phi(w)$ disjoint sets, each of size $\phi(w)$, and each of them congruent modulo w to $(\mathbb{Z}/w\mathbb{Z})^*$. Moreover, it is easy to see that when the order of 2 is the same in $(\mathbb{Z}/m\mathbb{Z})^*$ and $(\mathbb{Z}/w\mathbb{Z})^*$ then this partitioning can be made to respect the conjugacy classes of 2. Namely for any $t \in (\mathbb{Z}/w\mathbb{Z})^*$, we put $2t \bmod m$ in the same part as t. Such conjugation-respecting partition of $(\mathbb{Z}/m\mathbb{Z})^*$ can be constructed greedily, adding conjugacy classes from $(\mathbb{Z}/m\mathbb{Z})^*$ to the current part until we have a complete copy of $(\mathbb{Z}/w\mathbb{Z})^*$, then proceeding to the next part. Let $S_1, S_2, S_3, \ldots$ be this partition of $(\mathbb{Z}/m\mathbb{Z})^*$, so we have the properties:

- $S_i \cap S_j = \emptyset$ for all $i \neq j$, and $\cup_i S_i = (\mathbb{Z}/m\mathbb{Z})^*$;
- For all i we have $|S_i| = \phi(w)$, and also $S_i \bmod w = \{(s \bmod w) : s \in S_i\} = (\mathbb{Z}/w\mathbb{Z})^*$; and
- For all i we have $2S_i \bmod m = \{(2s \bmod m) \ : \ s \in S_i\} = S_i$.

Given the partition of $(\mathbb{Z}/m\mathbb{Z})^*$ to S_i's and the ordered representative set T_w for $\mathcal{Q}_w$, one way of getting an ordered representative set T_m for $\mathcal{Q}_m$ is to set

$$T_m \ = \ \left\{t \in (\mathbb{Z}/m\mathbb{Z})^* \ : \ \exists\, t' \in T_w \text{ s.t. } t \equiv t' \pmod{w}\right\},$$

obviously this set T_m has exactly one element from each conjugacy class in every part S_i. We can order it, $T_m = \{t_1, t_2, \ldots, t_{\ell_m}\}$, by taking all the elements from one part S_i before taking any of the elements from the next part S_{i+1}, and among the elements from the same part use the ordering of T_w.

Finally, fixing a specific primitive m'th root of unity $\zeta \in \mathbb{F}_{2^d}$ and the particular primitive w'th root of unity $\tau = \zeta^u$, we let the j'th plaintext slot encoded in $a \in R_{m,2}$ be the evaluation $a(\zeta^{t_j}) \in \mathbb{F}_{2^d}$, and similarly the j'th plaintext slot encoded in $a^* \in R_{w,2}$ is the evaluation $a^*(\tau^{t'_j})$. The following lemma plays an important role in our transformation:

Lemma 6. *Let $m = u \cdot w$ for odd integers u, w, and denote by d the order of 2 in $(\mathbb{Z}/m\mathbb{Z})^*$. Let ζ be a primitive m'th root of unity in $\mathbb{F}_{2^d}$, and denote $\tau = \zeta^u$, so τ is a primitive w'th root of unity.*

Let $S \subset (\mathbb{Z}/m\mathbb{Z})^*$ be a subset satisfying (a) $|S| = \phi(w)$ and $S \bmod w = (\mathbb{Z}/w\mathbb{Z})^*$, and (b) S is closed under multiplication by 2, $S = 2S \bmod m$. Then there exists a polynomial $h \in R_{w,2}$ such that for all $j \in S$, it holds that $h(\tau^j) = \zeta^j$.

Proof. Clearly, since $|S| = \phi(w)$ then there exists a unique polynomial h over $\mathbb{F}_{2^d}$ of degree smaller than $\phi(w)$ such that $h(\tau^j) = \zeta^j$ all $j \in S$. It is left to show only that h is a polynomial over the base field, i.e. with 0-1 coefficients. To show this, note that by definition of h we have $h(\tau^j) = \zeta^j$ for all $j \in S$, and moreover $2j \in S$ whenever $j \in S$ (and hence $h(\tau^{2j}) = \zeta^{2j}$). Thus, we get for all $j \in S$

$$h(\tau^{2j}) \;=\; \zeta^{2j} \;=\; (\zeta^j)^2 = h(\tau^j)^2.$$

Since $S \bmod w = (\mathbb{Z}/w\mathbb{Z})^*$ then the set $\{\tau^j : j \in S\}$ ranges over all the primitive w'th roots of unity in $\mathbb{F}_{2^d}$, so we have $h(\theta^2) = h(\theta)^2$ for *every primitive w'th root of unity θ*. It is a well-known fact that for an arbitrary polynomial $h(X)$ of degree smaller than $\phi(w)$ over $\mathbb{F}_{2^d}$, if $h(\theta^2) = h(\theta)^2$ holds for every primitive w'th root of unity $\theta \in \mathbb{F}_{2^d}$, then h is in fact a polynomial over the base field, i.e. a polynomial with 0-1 coefficients. This conclude the proof. □

We are now ready to show how to convert a big-ring ciphertext $\mathbf{c}$, encrypting some polynomial $a \in R_{m,2}$ into *a single small-ring ciphertext* that encrypt some other $a^* \in R_{w,2}$, such that a^* encodes all the plaintext elements that were encoded in the plaintext slots corresponding to one of the S_i's (i.e., all the slots $T_m \cap S_i$ for some S_i).

We begin by using the transformation from the previous section to construct from $\mathbf{c}$ the collection of u small-ring ciphertexts $\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{u-1}$ that encrypt the polynomials $\tilde{a}_0, \tilde{a}_1, \ldots, \tilde{a}_{u-1} \in R_{w,2}$, respectively, where the $\tilde{a}_k$'s are related to the original a via the assembly formula $a(X) = \sum_k X^k \cdot \tilde{a}_k(X^u) \bmod (\Phi_m, 2)$. Considering all of these 0-1 polynomials as members of $\mathbb{F}_{2^d}[X]$, and letting $\zeta \in \mathbb{F}_{2^d}$ be a primitive root of unity (so ζ is a root of $[\Phi_m \bmod 2]$ over $\mathbb{F}_{2^d}$), the assembly formula implies in particular that

$$a(\zeta^j) \;=\; \sum_{k=0}^{u-1} \zeta^{jk} \cdot \tilde{a}_k(\zeta^{ju}) \;=\; \sum_{k=0}^{u-1} \zeta^{jk} \cdot \tilde{a}_k(\tau^j) \quad \text{for every } j \in S_i$$

(where $\tau = \zeta^u$). Observing that S_i satisfies the conditions of Lemma 6, let $h \in R_{w,2}$ be the polynomial satisfying $h(\tau^j) = \zeta^j$ for all $j \in S_i$. Further, let us denote $h_k = (h^k \bmod (\Phi_w, 2)) \in R_{w,2}$. Since for all $j \in S_i$, τ^j is a primitive w'th root of unity (and hence a root of $[\Phi_w \bmod 2]$ over $\mathbb{F}_{2^d}$), then we get

$$h_k(\tau^j) \;=\; h(\tau^j)^k \;=\; \zeta^{jk} \;\text{ for every } j \in S_i.$$

We now set $\mathbf{c}^* = \sum_{k=0}^{u-1} h_k \cdot \mathbf{c}_k \bmod (\Phi_w, q)$, and note that this is a linear combination of the valid ciphertexts $\mathbf{c}_k$ with low-norm coefficients. (The h_k's have low norm because they are 0-1 polynomials.) Using the additive homomorphism of the cryptosystem (over the small ring R_w), this means that $\mathbf{c}^*$ is still a valid small-ring ciphertext, encrypting the polynomial $a^* = \sum_{k=0}^{u-1} h_k \cdot \tilde{a}_k \bmod (\Phi_w, 2) \in R_{w,2}$. Moreover, by our definition of the h_k's we have that for all $j \in T_m \cap S_i$,

$$a^*(\tau^j) \;=\; \sum_{k=0}^{u-1} h_k(\tau^j) \cdot \tilde{a}_k(\tau^j) \;=\; \sum_{k=0}^{u-1} \zeta^{jk} \cdot \tilde{a}_k(\zeta^{ju}) \;=\; a(\zeta^j).$$

Using our encoding conventions from the beginning of this section, this means that the content of the plaintext slots of a^* is exactly the content of the plaintext slots in a corresponding to $T_m \cap S_i$.

Ring-switching for "sparsely populated" ciphertexts. We mentioned that when the original big-ring ciphertext was sparsely populated, we would like to reduce it to only a small number of small-ring ciphertexts, only as many as needed to hold all the plaintext slots that contain real data. If the full slots are not already packed together in one (or a few) of the parts S_i, then we can apply the slot permutation techniques of Gentry et al. [12] to pack them as needed inside the big-ring ciphertext, before breaking it into the small-ring.

4.2 The General Case

The above treatment relies on the order of 2 in $(\mathbb{Z}/w\mathbb{Z})^*$ and $(\mathbb{Z}/m\mathbb{Z})^*$ being the same d. However, the only part that relies on this fact was Lemma 6, where we needed it in order to prove that the polynomial h is defined over the base field. In the general case this no longer holds, so although we can define the polynomials h_k (and therefore a^*) just as above, all of these polynomials will now have coefficients from the extension field $\mathbb{F}_{2^d}$ rather than 0-1 coefficients.[6] This is unavoidable in general, since we know that we cannot always encode $\mathbb{F}_{2^d}$ elements as polynomials in the small ring $R_{w,2}$.

In principle there is no problem with using plaintext arithmetic over $\mathbb{F}_{2^d}[X]/\Phi_w$ (rather than $R_{w,2} = \mathbb{F}_2[X]/\Phi(w)$). Fixing a representation $\mathbb{F}_{2^d} = \mathbb{F}_2[Y]/G(Y)$, we can represent the plaintext polynomial $A(X) \in \mathbb{F}_{2^d}[X]/\Phi_w(X)$ as a bivariate polynomial $A(X,Y) \in \mathbb{F}_2[X,Y]/(\Phi_w(X), G(Y))$, writing each coefficient from $\mathbb{F}_{2^d}$ as a degree-$(d-1)$ polynomial in Y. This means that A can be written as $A(X,Y) = \sum_{i=0}^{d-1} a_i(X)Y^i$ with the a_i's 0-1 polynomials in $R_{w,2}$. An encryption of a A then consists of d small-ring ciphertexts encrypting the a_i's, and arithmetic operations can be implemented naturally using our basic operations on encryptions of the a_i's. However, this is likely to be quite inefficient, probably even less efficient than keeping everything in the big ring.

We remark that in many settings, even though our plaintext slots can hold elements in $\mathbb{F}_{2^d}$, we really only use them to hold elements from a much smaller sub-field (e.g. bits or $\mathbb{F}_{2^8}$ elements). One could therefore hope that the technique from above could be generalized to map the $\mathbb{F}_{2^d}$ plaintext slots over the big ring into $\mathbb{F}_{2^{d'}}$ slots over the small ring, such that if the content of the slots happened to already belong to the subfield $\mathbb{F}_{2^{d'}}$ then it will be copied intact. Finding such a generalization for every $d'|d$ is an interesting open problem.

For the case where we use the plaintext slots to hold just bits, it turns out that we can use a slight adaptation of the procedure for $d' = d$. In this case, the transformation from above yields an encryption of a polynomial $A(X)$ over $\mathbb{F}_{2^d}$, that contains in its slots whatever we had in the original big-ring polynomial. In particular it means that $A(\tau^k) \in \{0,1\}$ for every k, hence in this case *A must be a 0-1 polynomial*. So after we compute an encryption of A (as a set of d encryptions as above), we can just discard all the ciphertexts except the one corresponding to a_0.

References

[1] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 595–618. Springer, 2009.

[2] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In *ICALP (1)*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011.

[3] Sanjeev Arora and Rong Ge. New algorithms for learning in the presence of errors. Manuscript, 2011.

[4] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. Manuscript available at `http://eprint.iacr.org/2012/078`.

[5] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science (ITCS'12)*, 2012. Available at `http://eprint.iacr.org/2011/277`.

[6] Sometimes it is possible to show that the coefficients are drawn from a smaller extension field.

[6] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[7] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS'11*. IEEE Computer Society, 2011.

[8] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. Available at `http://eprint.iacr.org/2011/535`.

[9] Leo Ducas and Alain Durmus. Ring-LWE in Polynomial Rings. To appear in PKC 2012, manuscript availabale from `http://eprint.iacr.org/2012/235`

[10] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 2008.

[11] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 446-464, 2012. Full version at `http://eprint.iacr.org/2011/566`, 2012.

[13] Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping for fully homomorphic encryption. To appear *PKC 2012*. `http://eprint.iacr.org/2011/680`, 2011.

[14] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. Manuscript, 2012.

[15] Adriana Lòpez-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption In *STOC* 2012.

[16] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, 2010.

[17] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for Ring-LWE cryptography. Manuscript, 2012

[18] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, 2011.

[19] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. Manuscript at http://eprint.iacr.org/2011/133, 2011.

Field Switching in BGV-Style Homomorphic Encryption

Craig GENTRY [a] Shai HALEVI [a] Chris PEIKERT [b] and Nigel P. SMART [c]
[a] *IBM T.J. Watson Research Center*
[b] *Georgia Institute of Technology*
[c] *University of Bristol*

Abstract. The security of contemporary homomorphic encryption schemes over cyclotomic number field relies on fields of very large dimension. This large dimension is needed because of the large modulus-to-noise ratio in the key-switching matrices that are used for the top few levels of the evaluated circuit. However, a smaller modulus-to-noise ratio is used in lower levels of the circuit, so from a security standpoint it is permissible to switch to lower-dimension fields, thus speeding up the homomorphic operations for the lower levels of the circuit. However, implementing such field-switching is nontrivial, since these schemes rely on the field algebraic structure for their homomorphic properties.

A basic ring-switching operation was used by Brakerski, Gentry and Vaikuntanathan, over rings of the form $\mathbb{Z}[X]/(X^{2^n}+1)$, in the context of bootstrapping. In this work we generalize and extend this technique to work over any cyclotomic number field, and show how it can be used not only for bootstrapping but also during the computation itself (in conjunction with the "packed ciphertext" techniques of Gentry, Halevi and Smart).

Keywords. Homomorphic Encryption, Ring-LWE

1. Introduction

The last few years have seen a rapid advance in the state of fully homomorphic encryption, yet despite these advances, the existing schemes are still too expensive for many practical purposes. In this paper we make another step forward in making such schemes more efficient. In particular, we present a technique for reducing the dimension of the ciphertexts involved in the homomorphic computation of the lower levels of a circuit. Our techniques apply to homomorphic encryption schemes over number fields, such as the schemes of Brakerski et al. [4,5,3], as well as the variants due to López-Alt et al. [14] and Brakerski [2].

The most efficient variants of these schemes work over number fields of the form $\mathbb{Q}(\zeta) \cong \mathbb{Q}[X]/F(X)$, and in all of them the field dimension n, which is the degree of $F(X)$, must be set large enough to ensure security: to support homomorphic evaluation of depth-L circuits with security parameter λ, the schemes require $n = \tilde{\Omega}(L \cdot \text{polylog}(\lambda))$, even under the strongest plausible hardness assumptions

for their underlying computational problems (e.g., ring-LWE [15]).[1] In practice, the field dimension for moderately deep circuits can easily be many thousands. For example, to be able to evaluate AES homomorphically, Gentry et al. [13] used circuits of depth $L \geq 50$, with a corresponding field dimension of over 50,000.

As homomorphic operations are performed, the ratio of noise to modulus in the ciphertexts grows. Consequently, it becomes permissible to use lower-dimension fields, which can speed up further homomorphic computations. However, since we must start with ciphertexts from a high-dimensional field, we need a method for transforming them into small-field ciphertexts that encrypt the same (or related) messages. Such a "field switching" procedure was described by Brakerski et al. [3], in the context of reducing the ciphertext size prior to bootstrapping. The procedure in [3], however, is specific to number fields of the form $K_{2^k} = \mathbb{Q}[X]/(X^{2^{k-1}} + 1)$, i.e., cyclotomic number fields with power-of-2 index. Moreover, by itself it cannot be combined with the "packed evaluation" techniques from [18,11]. (These techniques use Chinese-remainder encoding to "pack" many plaintext values into each ciphertext, and then each homomorphic operation is applied to all these values at once. For our purposes, we must consider the effect of the field-switching operation on all these plaintext values.) Extending and improving the field switching procedure is the goal of our work.

1.1. Our Contribution

We present a general field-switching transformation that can be applied to any *cyclotomic* number field $K = \mathbb{Q}(\zeta_m) \cong \mathbb{Q}[X]/\Phi_m(X)$ for arbitrary m (where $\Phi_m(X) \in \mathbb{Z}[X]$ is the mth cyclotomic polynomial), and works well in conjunction with packed ciphertexts. For any divisor m' of m, our procedure takes as input a "big-field ciphertext" c over K that encrypts many plaintext values, and outputs a "small-field ciphertext" c' over $K' = \mathbb{Q}(\zeta_{m'}) \cong \mathbb{Q}[X]/\Phi_{m'}(X) \subseteq K$ that encrypts a certain subset of the input plaintext values.[2]

Our transformation relies heavily on the algebraic properties of the cyclotomic number fields K, K' and their respective rings of algebraic integers R, R'. In particular, we use the interpretation of K as an extension field of K', and relationships between their various embeddings into the complex numbers $\mathbb{C}$; the factorization of integer primes in R and R'; and the *trace function* $\mathrm{Tr}_{K/K'}$ that maps elements in K to the subfield K'. With these tools in hand, the transformation itself is quite simple, and consists of the following three steps:

1. We first apply a key-switching operation to obtain a big-field ciphertext over K with respect to a small-field secret key $s' \in K' \subset K$. Proving the security of this operation relies on a novel way of embedding the ring-LWE problem over K' into K, which may be of independent interest.

[1] The schemes from [3,2] can also obtain security by using high-dimensional vectors over low-dimensional number fields. But their most efficient variants use low-dimensional vectors over high-dimensional fields, since the runtime of certain operations is cubic in the dimension of the vectors.

[2] More generally, the output ciphertext can even encrypt certain linear functions of the input plaintext values.

2. Next, we multiply the resulting ciphertext by a certain element of the ring $R \subset K$, which depends only on the subset (or other function) of the plaintext values that we want to include in the output ciphertext.
3. Finally, we take the trace of the K-elements in the ciphertext, thus obtaining an output ciphertext over the subfield K', which decrypts under the secret key $s' \in K'$ to the desired plaintext values.

We note that in addition to being simpler and more general than the transformation from [3], our transformation is also more efficient even when applied in the special case of K_{2^k}: when switching from K_{2^k} to $K_{2^{k'}}$, the transformation from [3] includes a step where the size of the ciphertext (and hence the time that it takes to perform operations) is expanded by a factor of $2^{k-k'}$. Our transformation does not need that extra step, hence saving this extra factor in performance.

In Section 2 below we recall the algebraic concepts needed for our transformation, and then the transformation itself it described in Section 3.

2. Preliminaries

This work uses a number of algebraic concepts and notations; to assist the reader we summarize the most important ones in Table 1. For any positive integer u we let $[u] = \{0, \ldots, u-1\}$. Throughout this work, for a coset $z \in \mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ we let $[z]_q \in \mathbb{Z}$ denote its canonical representative in $\mathbb{Z} \cap [-q/2, q/2)$. One can also view $[\cdot]_q$ as the operation that takes an arbitrary integer z and reduces it modulo q into the interval $[-q/2, q/2)$.

2.1. Algebraic Background

Recall that an *ideal* I in a commutative ring R is a nontrivial (i.e., $I \neq \emptyset$ and $I \neq \{0\}$) additive subgroup which is closed under multiplication by R. For ideals I, J, their sum is the ideal $I + J = \{a + b : a \in I, b \in J\}$, and their product IJ is the ideal consisting of all *sums* of terms ab for $a \in I, b \in J$. An R-ideal $\mathfrak{p}$ is prime if $ab \in \mathfrak{p}$ (for some $a, b \in R$) implies $a \in \mathfrak{p}$ or $b \in \mathfrak{p}$ (or both). All the rings we work with have unique factorization of ideals into powers of prime ideals, and a Chinese Remainder Theorem.

A *fractional ideal* is, informally, an ideal with a denominator. Formally, letting K be the field of fractions of R, a fractional ideal of R is a subset $I \subseteq K$ for which there exists a denominator $d \in R$ such that $dI \subseteq R$ is an ideal in R. For an R-ideal I, the quotient ring $R_I = R/I$ consists of the residue classes $a + I$ for all $a \in R$, with the ring operations induced by R. More generally, for a (possibly fractional) ideal I and an ideal $J \subseteq R$, the quotient $I_J = I/IJ$ is an additive group, and an R-module, with addition and multiplication operations induced by R. We often write $a \bmod I$ instead of $a + I$ to denote the residue classes $a + I$, and we write $a = b \pmod{*}I$ to denote that a, b belong to the same residue class, i.e., $a + I = b + I$.

For computational purposes, all of the rings and fields we work with have efficient representations of their elements, and efficient (i.e., polynomial time in the bit length of the arguments) algorithms for all the operations we use. For quotients

Notations	Description
p, $\mathbb{F}_{p^d}$	The (prime) modulus of the cryptosystem's native plaintext space, and the finite field of order p^d.
m, m', $n = \varphi(m), n' = \varphi(m')$	The indices of the cyclotomic fields, where $m'\|m$. We switch from the mth to the m'th cyclotomic number field, which are of degree n, n' (respectively) over the rationals.
$\bar{m},\ d,\ e,\ f$, $\bar{m}', d', e', f'$	$\bar{m}$ is the largest divisor of m that is coprime with p; d is the order of p in $\mathbb{Z}^*_{\bar{m}}$; $e = \varphi(m)/\varphi(\bar{m})$; and $f = \varphi(\bar{m})/d$. Similarly for $\bar{m}', d', e', f'$.
ζ_m, $\zeta_{m'}$	Abstract elements of order m, m' (respectively) over the rationals.
$K = \mathbb{Q}(\zeta_m)$, $K' = \mathbb{Q}(\zeta_{m'})$, $R = \mathbb{Z}[\zeta_m]$, $R' = \mathbb{Z}[\zeta_{m'}]$	The cyclotomic number fields and their rings of integers.
$\sigma\colon K \to \mathbb{C}^n$ $\sigma'\colon K' \to \mathbb{C}^{n'}$	The canonical embeddings of K, K', which endow the number fields with a geometry.
$\mathrm{Tr}_{K/K'}\colon K \to K'$	The trace function, which is the sum of the automorphisms of K that fix K' pointwise.
$R^\vee, (R')^\vee$	The codifferent (or dual) fractional ideals of R and R' (respectively), defined as $R^\vee = \{a : \mathrm{Tr}_{K/\mathbb{Q}}(aR) \subseteq \mathbb{Z}\}$ and similarly for $(R')^\vee$.
$G = \mathbb{Z}^*_{\bar{m}}/\langle p\rangle$, $G' = \mathbb{Z}^*_{\bar{m}'}/\langle p\rangle$	The multiplicative quotient groups that characterize the prime-ideal factorizations of pR, pR', respectively.
$g\colon G \to G'$	The (f/f')-to-1 homomorphism defined via $i \mapsto i \bmod \bar{m}'$.

Table 1. Summary of the main algebraic notations.

A/B, cosets are represented using a fixed set of distinguished representatives. In this work we largely ignore the details of concrete representations and algorithms, and refer to [16] for fast, specialized algorithms for working with the cyclotomic fields and rings that we use in this work.

2.1.1. Cyclotomic Fields and Rings

For a positive integer m, let $K = \mathbb{Q}(\zeta_m)$ be the mth *cyclotomic number field*, where ζ_m is an abstract element of order m. (In particular, we do not view ζ_m as any particular root of unity in $\mathbb{C}$.) The minimal polynomial of ζ_m is the mth *cyclotomic polynomial* $\Phi_m(X) = \prod_{i\in\mathbb{Z}^*_m}(X - \eta_m^i) \in \mathbb{Z}[X]$, where $\eta_m = \exp(2\pi\sqrt{-1}/m) \in \mathbb{C}$ is the principal mth complex root of unity, and the roots $\eta_m^i \in \mathbb{C}$ range over all the *primitive* complex mth roots of unity. Therefore, K is a field extension of degree $n = \varphi(m)$ over $\mathbb{Q}$, and is isomorphic to the polynomial ring $\mathbb{Q}[X]/\Phi_m(X)$ by identifying ζ_m with X. (There are other representations of K as well, and nothing in this work depends on a particular choice of representation.) The *ring of (algebraic) integers* in K, called the mth cyclotomic ring, is $R = \mathbb{Z}[\zeta_m]$, which is isomorphic to $\mathbb{Z}[X]/\Phi_m(X)$.

The field extension $K/\mathbb{Q}$ has n automorphisms $\tau_i\colon K \to K$ that fix $\mathbb{Q}$ pointwise, which are characterized by $\tau_i(\zeta_m) = \zeta_m^i$ for $i \in \mathbb{Z}^*_m$. (Equivalently, $\tau_i(a(X)) =$

$a(X^i) \bmod \Phi_m(X)$ when viewing K as $\mathbb{Q}[X]/\Phi_m(X)$.) Because $K/\mathbb{Q}$ is Galois (i.e., the number of automorphisms equals the dimension of the extension), the $\mathbb{Q}$-linear[3] *(field) trace* $\mathrm{Tr}_{K/\mathbb{Q}}\colon K \to \mathbb{Q}$ can be defined as the sum of the automorphisms: $\mathrm{Tr}_{K/\mathbb{Q}}(a) = \sum_{i\in\mathbb{Z}_m^*} \tau_i(a) \in \mathbb{Q}$. (See below for another formulation.)

Similarly to the automorphisms τ_i (which map K to itself), there are n concrete ways of viewing K as a subfield of the complex numbers $\mathbb{C}$. Namely, there are n injective ring homomorphisms from K to $\mathbb{C}$ that fix $\mathbb{Q}$ pointwise, called *embeddings*, which are denoted $\sigma_i\colon K \to \mathbb{C}$ for $i \in \mathbb{Z}_m^*$ and characterized by $\sigma_i(\zeta_m) = \eta_m^i$. The embeddings may be seen as the compositions of the abstract automorphisms τ_i with the complex embedding that identifies $\zeta_m \in K$ with $\eta_m \in \mathbb{C}$. Therefore, the field trace can also be written as the sum of the embeddings, as $\mathrm{Tr}_{K/\mathbb{Q}}(a) = \sum_{i\in\mathbb{Z}_m^*} \sigma_i(a) \in \mathbb{Q}$. The *canonical embedding* $\sigma\colon K \to \mathbb{C}^n$ is the concatenation of all the complex embeddings, i.e., $\sigma(a) = (\sigma_i(a))_{i\in\mathbb{Z}_m^*}$, and it endows K with a canonical geometry. In particular, define the Euclidean (ℓ_2) and ℓ_∞ norms on K as

$$\|a\| := \|\sigma(a)\| = \sqrt{\sum_i |\sigma_i(a)|^2} \quad \text{and} \quad \|a\|_\infty := \|\sigma(a)\|_\infty = \max_i |\sigma_i(a)|,$$

respectively. Note that $\|a \cdot b\| \le \|a\|_\infty \cdot \|b\|$ and $\|a \cdot b\|_\infty \le \|a\|_\infty \cdot \|b\|_\infty$ for any $a, b \in K$, because the σ_i are ring homomorphisms.

2.1.2. Towers of Cyclotomics

For any positive integer m' dividing m, let $K' = \mathbb{Q}(\zeta_{m'})$ and $R' = \mathbb{Z}[\zeta_{m'}]$ be the m'th cyclotomic field and ring (of dimension $n' = \varphi(m')$ over $\mathbb{Q}$ and $\mathbb{Z}$), respectively. As above, the field extension $K'/\mathbb{Q}$ has $n' = \varphi(m')$ automorphisms $\tau'_{i'}\colon K' \to K'$ and n' complex embeddings $\sigma'_{i'}\colon K' \to \mathbb{C}$ (for $i' \in \mathbb{Z}_{m'}^*$), the latter of which define the canonical embedding $\sigma'\colon K' \to \mathbb{C}^{n'}$.

We will use extensively the fact that K is a field extension of K', and R is a ring extension of R', both of dimension n/n' (because $K/\mathbb{Q}$ and $K'/\mathbb{Q}$ have dimensions n and n', respectively). That is, K' and R' may respectively be seen as a subfield of $K = K'(\zeta_m)$ and a subring of $R = R'[\zeta_m]$, under the ring embedding that identifies $\zeta_{m'}$ with $\zeta_m^{m/m'}$. Moreover, the field extension K/K' is Galois, i.e., it has n/n' automorphisms that fix K' pointwise, which are precisely those τ_i for which $i = 1 \pmod{*}m'$. This follows from the fact that

$$\tau_i(\zeta_{m'}) = \tau_i(\zeta_m^{m/m'}) = \zeta_m^{(m/m')i \bmod m} = \zeta_{m'}^{i \bmod m'}, \tag{1}$$

and that reducing modulo m' induces an (n/n')-to-1 mapping from $\mathbb{Z}_m^*$ to $\mathbb{Z}_{m'}^*$. The K'-linear (intermediate) trace function $\mathrm{Tr}_{K/K'}\colon K \to K'$ may be defined as the sum of these automorphisms:

$$\mathrm{Tr}_{K/K'}(a) = \sum_{i=1 \pmod{*}m'} \tau_i(a).$$

[3] A function f is S-linear if $f(a+b) = f(a) + f(b)$ and $f(s \cdot a) = s \cdot f(a)$ for all $s \in S$ and all a, b.

A standard fact from field theory is that the intermediate trace satisfies $\mathrm{Tr}_{K/\mathbb{Q}} = \mathrm{Tr}_{K'/\mathbb{Q}} \circ \mathrm{Tr}_{K/K'}$. Another standard fact is that $\mathrm{Tr}_{K/K'}$ is a "universal" K'-linear function, in that any such function $L\colon K \to K'$ can be expressed as $L(a) = \mathrm{Tr}_{K/K'}(r \cdot a)$ for some fixed $r \in K$.

Similarly to Equation (1), for any $i \in \mathbb{Z}_m^*$ the embedding σ_i coincides with $\sigma'_{i \bmod m'}$ on the subfield K'. Using this fact we get the following relation between the intermediate trace and the complex embeddings of K and K'.

Lemma 2.1. *For any $a \in K$ and $i' \in \mathbb{Z}_{m'}^*$,*

$$\sigma'_{i'}(\mathrm{Tr}_{K/K'}(a)) = \sum_{i=i' \pmod{*m'}} \sigma_i(a).$$

In matrix form, $\sigma'(\mathrm{Tr}_{K/K'}(a)) = P \cdot \sigma(a)$, where P is the $\varphi(m')$-by-$\varphi(m)$ matrix (with rows indexed by $i' \in \mathbb{Z}_{m'}^*$ and columns by $i \in \mathbb{Z}_m^*$) whose (i', i)th entry is 1 if $i = i' \pmod{*m'}$, and is 0 otherwise.

Proof. Fix an arbitrary $k \in \mathbb{Z}_m^*$ such that $k = i' \pmod{*m'}$. Then because $\sigma'_{i'}$ coincides with σ_k on K', and by definition of $\mathrm{Tr}_{K/K'}$ and linearity of σ_k, we have

$$\begin{aligned}\sigma'_{i'}(\mathrm{Tr}_{K/K'}(a)) = \sigma_k([)\Big]_{j=1} & \sum_{(\mathrm{mod}\ *)m'} \tau_j(a) \\ &= \sum_{j=1 \pmod{m'}} \sigma_k(\tau_j(a)) = \sum_{i=i' \pmod{m'}} \sigma_i(a),\end{aligned}$$

where for the last equality we have used $\sigma_k \circ \tau_j = \sigma_{k \cdot j}$ and $k \in \mathbb{Z}_m^*$, so $i = k \cdot j \in \mathbb{Z}_m^*$ runs over all indices congruent to i' modulo m' when $j \in \mathbb{Z}_m^*$ runs over all indices congruent to 1 modulo m'. □

An immediate corollary is that the intermediate trace maps short elements of K to short elements of K'.

Corollary 2.2. *For any $a \in K$, we have $\|\mathrm{Tr}_{K/K'}(a)\| \le \|a\| \cdot \sqrt{n/n'}$.*

Proof. By Lemma 2.1, we have $\sigma'(\mathrm{Tr}_{K/K'}(a)) = P \cdot \sigma(a)$. The rows of P are orthogonal (since each column of P has exactly one nonzero entry), and each has Euclidean norm exactly $\sqrt{n/n'}$. □

2.1.3. Prime Splitting and Plaintext Arithmetic

We now describe the factorization ("splitting") of prime integers in cyclotomic rings, how it allows for encoding and operating on several finite-field elements, and the particular functions induced by the (intermediate) trace function $\mathrm{Tr}_{K/K'}$. Further details and proofs can be found in many texts on algebraic number theory, e.g., [19].

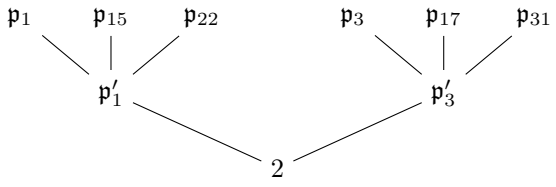

Figure 1. Factorization of $2 \in \mathbb{Z}$ into distinct prime ideals $\mathfrak{p}'_{i'}$ in $R' = \mathbb{Z}[\zeta_7]$, and $\mathfrak{p}_i$ in $R = \mathbb{Z}[\zeta_{91}]$. The displayed subscripts indicate a choice of representatives from the cosets of the multiplicative subgroups $\langle 2 \rangle \subseteq \mathbb{Z}_7^*$ and $\langle 2 \rangle \subseteq \mathbb{Z}_{91}^*$, which have orders $d' = 3$ and $d = 12$, respectively.

Prime splitting. Let $p \in \mathbb{Z}$ be a prime integer. In the mth cyclotomic ring $R = \mathbb{Z}[\zeta_m]$ (which has degree $n = \varphi(m)$ over $\mathbb{Z}$), pR is often not a prime ideal, but instead factors into prime ideals. To describe how, we first need to introduce some notation. Divide out all the factors of p from m, writing $m = \bar{m} \cdot p^k$ where $p \nmid \bar{m}$. Let $e = \varphi(p^k)$, and let d be the multiplicative order of p modulo $\bar{m}$ (i.e., in $\mathbb{Z}_{\bar{m}}^*$); note that d divides $\varphi(\bar{m}) = n/e$. (The values d, e are respectively called the *inertial degree* and *ramification index* of p in R.) Let $G = \mathbb{Z}_{\bar{m}}^*/\langle p \rangle$, the multiplicative quotient group $\mathbb{Z}_{\bar{m}}^*$ modulo the order-d subgroup generated by p, so G has order $f = \varphi(\bar{m})/d = n/(de)$. For an element $i \in G$ of this group, we sometimes write $i\langle p \rangle$ to emphasize that it is a coset, and (slightly abusing notation) also let $i \in \mathbb{Z}_{\bar{m}}^*$ denote some element of the coset. The ideal pR factors as

$$pR = \prod_{i \in G} \mathfrak{p}_i^e, \tag{2}$$

where the $\mathfrak{p}_i$ are distinct prime ideals in R, all having norm $|R/\mathfrak{p}_i| = p^d$. These are called the prime ideals *lying over* p in R. Each quotient ring $R/\mathfrak{p}_i$ is therefore isomorphic to the finite field $\mathbb{F}_{p^d}$. (In fact there are exactly d isomorphisms between them, because $\mathbb{F}_{p^d}$ has d automorphisms.)

Concretely, the prime ideals $\mathfrak{p}_i$, and the isomorphisms between $R/\mathfrak{p}_i$ and (some canonical representation of) $\mathbb{F}_{p^d}$, are as follows. Let $\omega_{\bar{m}}$ denote some arbitrary element of order $\bar{m}$ in $\mathbb{F}_{p^d}$; such an element exists because the multiplicative group $\mathbb{F}_{p^d}^*$ is cyclic and has order $p^d - 1 = 0 \pmod{*}\bar{m}$. For any $i\langle p \rangle \in G$, the prime ideal $\mathfrak{p}_i$ is the kernel of the ring homomorphism $h_i \colon R \to \mathbb{F}_{p^d}$ defined by $h_i(\zeta_m) = \omega_{\bar{m}}^i$. It is immediate that this kernel is an ideal; furthermore, it is invariant under the choice of representative i from the coset $i\langle p \rangle$, because $h_{ip}(r) = h_i(r)^p$ for any $r \in R$ (since $(a+b)^p = a^p + b^p$ for any $a, b \in \mathbb{F}_{p^d}$). Because $\mathfrak{p}_i$ is the kernel of h_i, we have the induced isomorphism $h_i \colon R/\mathfrak{p}_i \to \mathbb{F}_{p^d}$; indeed, we have d distinct such isomorphisms, one for each element of the coset $i\langle p \rangle$.

Looking ahead, the isomorphisms h_i (for appropriate choices of representatives i) will be used to define several "plaintext slots" in a homomorphic cryptosystem, i.e., an encoding of f plaintext elements of $\mathbb{F}_{p^d}$ as a single element of the cryptosystem's plaintext ring $R/2R$.

Splitting in cyclotomic towers. Of course, the above derivation also applies to the ideals that lie over p in $R' = \mathbb{Z}[\zeta_{m'}] \subseteq R$. For each such ideal $\mathfrak{p}'$, we next describe the factorization of $\mathfrak{p}'R$ into prime ideals in R. These are the prime ideals that lie

over $\mathfrak{p}'$ in R, and since "lying over" is an associative property, they also lie over p (as illustrated in Figure 1).

Let $\bar{m}, d, e, f, G$ and the prime ideals $\mathfrak{p}_i$ for $i \in G$ be as above for R, and define $\bar{m}', d', e', f', G' = \mathbb{Z}^*_{\bar{m}'}/\langle p \rangle$ and prime ideals $\mathfrak{p}'_{i'}$ for $i' \in G'$ similarly for R'. Note that $d'|d$, $e'|e$, and $f'|f$, and that the natural homomorphism $g\colon G \to G'$ defined via $i \mapsto i \bmod \bar{m}'$ is surjective and (f/f')-to-1. Then for every $i' \in G'$, the factorization of $\mathfrak{p}'_{i'}R$ is

$$\mathfrak{p}'_{i'}R = \prod_{i \in g^{-1}(i')} \mathfrak{p}_i^{e/e'} = \prod_{i = i' \bmod \bar{m}} \mathfrak{p}_i^{e/e'}.$$

Therefore, there are f/f' prime ideals of R lying over each $\mathfrak{p}'_{i'}$, and taken over all $i' \in G'$ they partition the prime ideals of R lying over p.

Plaintext encoding. Let $\mathbb{F} = \mathbb{F}_{p^d}$ and $\mathbb{F}' = \mathbb{F}_{p^{d'}} \subseteq \mathbb{F}$. By the above and the Chinese Remainder Theorem, the natural ring homomorphisms yield the following (where $\cong$ denotes a ring isomorphism):

$$R'/pR' \to R'/([]) \prod_{i' \in G'} \mathfrak{p}'_{i'} \cong \bigoplus_{i' \in G'} R'/\mathfrak{p}'_{i'} \cong \mathbb{F}'^{f'}$$

$$R/pR \to R/([]) \prod_{i \in G} \mathfrak{p}_i = R/([]) \prod_{i' \in G'} \prod_{i \in g^{-1}(i')} \mathfrak{p}_i \cong \bigoplus_{i' \in G'} \bigoplus_{i \in g^{-1}(i')} R/\mathfrak{p}_i \cong (\mathbb{F}^{f/f'})^{f'}.$$

(Note that the first homomorphism in each line is surjective, but not necessarily an isomorphism, due to possible ramification.) Following [18,3,11,12,13], in the context of homomorphic encryption the above morphisms allow for encoding a vector of f' individual elements of $\mathbb{F}'$ (respectively, f elements of $\mathbb{F}$) into the plaintext ring $R'_p = R/pR'$ (resp., $R_p = R/pR$), so that a single homomorphic addition and multiplication acts component-wise on the underlying vectors of field elements.

Trace operations. As mentioned in the introduction, our field-switching technique is built around applying the trace function $\mathrm{Tr}_{K/K'}$ to the elements of a big-field ciphertext, thus obtaining a related small-field ciphertext. Since we use "packed" ciphertexts that encrypt arrays of elements in $\mathbb{F}$ via the above isomorphisms, we need to understand the effect of the trace function on those $\mathbb{F}$-elements.

The remainder of this subsection is therefore devoted to characterizing the functions $(\mathbb{F}^{f/f'})^{f'} \to \mathbb{F}'^{f'}$ that can be induced by $\mathrm{Tr}_{K/K'}$. More specifically, we determine exactly which functions

$$L\colon R/(\prod_{i \in G} \mathfrak{p}_i) \to R'/(\prod_{i' \in G'} \mathfrak{p}'_{i'})$$

can be expressed as $L(a) = \mathrm{Tr}_{K/K'}(r \cdot a)$ for some fixed $r \in K$. It turns out that by fixing an appropriate choice of isomorphisms between the quotient rings and

finite fields above, we can obtain the *concatenation* of any f' individual $\mathbb{F}'$-linear functions $\mathbb{F}^{f/f'} \to \mathbb{F}'$ (see Corollary 2.5 for a precise statement).[4]

As already noted, the isomorphisms between the quotient rings and finite fields are not necessarily unique; they are determined by the choice of representatives i', i of the cosets $i'\langle p\rangle \subseteq \mathbb{Z}^*_{\bar{m}'}$ and $i\langle p\rangle \subseteq \mathbb{Z}^*_{\bar{m}}$ (respectively), and roots of unity $\omega_{\bar{m}'} \in \mathbb{F}'$ and $\omega_{\bar{m}} \in \mathbb{F}$. For our purposes, it is important to choose these in a "consistent" fashion, as follows. First, given $\omega_{\bar{m}}$, let $\omega_{\bar{m}'} = \omega_{\bar{m}}^{\bar{m}/\bar{m}'} \in \mathbb{F}'$. (Note that all $\varphi(\bar{m}')$ elements of order $\bar{m}'$ in $\mathbb{F}$ are indeed in the subfield $\mathbb{F}'$.) Next, let $\ell \geq 0$ be the integer exponent such that $m/m' = (\bar{m}/\bar{m}') \cdot p^\ell$. Then given representative i' of $i'\langle p\rangle \in G'$, choose representative i for each $i\langle p\rangle \in g^{-1}(i')$ so that $p^\ell \cdot i = i' \pmod{*}\bar{m}'$. Note that such i always exists, by definition of the quotient group G and the mapping g. As explained above, these choices fix particular isomorphisms

$$h_i \colon R/\mathfrak{p}_i \to \mathbb{F} \text{ (for } i\langle p\rangle \in G) \quad \text{and} \quad h'_{i'} \colon R'/\mathfrak{p}'_{i'} \to \mathbb{F}' \text{ (for } i'\langle p\rangle \in G'),$$

which are characterized by $h_i(\zeta_m) = \omega_{\bar{m}}^i$ and $h'_{i'}(\zeta_{m'}) = \omega_{\bar{m}'}^{i'}$.

Next, for each $i' \in G'$ denote the product of prime ideals lying over $\mathfrak{p}'_{i'}$ in R (called the *radical* of $\mathfrak{p}'_{i'}R$) by $\tilde{\mathfrak{p}}_{i'} = \prod_{i \in g^{-1}(i')} \mathfrak{p}_i$, and define the ring isomorphism

$$\tilde{h}_{i'} \colon R/\tilde{\mathfrak{p}}_{i'} \to \mathbb{F}^{f/f'}, \quad \tilde{h}_{i'}(a) = ([])\big] h_i(a \bmod \mathfrak{p}_i)_{i \in g^{-1}(i')},$$

where $\mathbb{F}^{f/f'}$ denotes the product ring with coordinate-wise operations.

In Lemma 2.4 below, we show that under the above isomorphisms, the $\mathbb{F}'$-linear functions $\bar{L} \colon \mathbb{F}^{f/f'} \to \mathbb{F}'$ correspond bijectively with the R'-linear functions $L \colon R/\tilde{\mathfrak{p}}_{i'} \to R'/\mathfrak{p}'_{i'}$, for all $i' \in G'$. Recall that any function of the latter type can be expressed as $L(a) = \mathrm{Tr}_{K/K'}(r \cdot a)$ for some fixed $r \in K$. Conversely, every function L (with domain and range as above) that can be expressed as $L(a) = \mathrm{Tr}_{K/K'}(r \cdot a)$ is clearly R'-linear, so it always induces an $\mathbb{F}'$-linear function. The heart of Lemma 2.4 is the following fact.

Lemma 2.3. *Let $\mathfrak{p}'_{i'}$ for some $i' \in G'$ be a prime ideal lying over p in R', and let $\tilde{\mathfrak{p}}_{i'}$ be the radical of $\mathfrak{p}_{i'}R$. Let $r' \in R' \subseteq R$ be arbitrary, and let $s = h'_{i'}(r' \bmod \mathfrak{p}'_{i'}) \in \mathbb{F}' \subseteq \mathbb{F}$. Then*

$$\tilde{h}_{i'}(r' \bmod \tilde{\mathfrak{p}}_{i'}) = (s, s, \ldots, s) \in \mathbb{F}'^{f/f'},$$

i.e., every entry of $\tilde{h}_{i'}(r' \bmod \tilde{\mathfrak{p}}_{i'})$ is equal to $h'_{i'}(r' \bmod \mathfrak{p}'_{i'})$.

Proof. Recall that under our choice of isomorphisms, $\omega_{\bar{m}'} = \omega_{\bar{m}}^{\bar{m}/\bar{m}'} \in \mathbb{F}'$ is of order $\bar{m}'$, and $p^\ell \cdot i = i' \bmod \bar{m}'$, where $\ell \geq 0$ is the integer satisfying $m/m' = (\bar{m}/\bar{m}') \cdot p^\ell$. Also recall that

$$\tilde{h}_{i'}(r' \bmod \tilde{\mathfrak{p}}_{i'}) = \big(h_i(r' \bmod \mathfrak{p}_i)\big)_{i \in g^{-1}(i')}.$$

[4]Note that any $\mathbb{F}'$-linear function $L \colon \mathbb{F}^{f/f'} \to \mathbb{F}'$ can always be expressed as $L(\vec{a}) = \mathrm{Tr}_{\mathbb{F}/\mathbb{F}'}(\langle \vec{d}, \vec{a}\rangle)$ for some fixed $\vec{d} \in \mathbb{F}^{f/f'}$, where $\langle \cdot, \cdot \rangle$ is the usual inner product and $\mathrm{Tr}_{\mathbb{F}/\mathbb{F}'}$ denotes the ($\mathbb{F}'$-linear) trace of the field extension $\mathbb{F}/\mathbb{F}'$.

For the representative i of each coset $i\langle p\rangle \in g^{-1}(i')$, the entry $h_i(r' \bmod \mathfrak{p}_i)$ is obtained by mapping ζ_m to $\omega_{\bar{m}}^i$, and hence also mapping $\zeta_{m'} = \zeta_m^{m/m'} = \zeta_m^{(\bar{m}/\bar{m}')\cdot p^\ell}$ to

$$\omega_{\bar{m}}^{(\bar{m}/\bar{m}')\cdot p^\ell \cdot i} = \omega_{\bar{m}'}^{p^\ell \cdot i} = \omega_{\bar{m}'}^{i'} \in \mathbb{F}',$$

which is exactly the mapping done by $h'_{i'}$. Since $r' \in R' = \mathbb{Z}[\zeta_{m'}]$, this proves the claim. □

Lemma 2.4. *Let $i' \in G'$ be arbitrary, and let $\mathfrak{p}' = \mathfrak{p}'_{i'}$ and $\tilde{\mathfrak{p}} = \tilde{\mathfrak{p}}_{i'}$. Then under the isomorphisms $h' = h'_{i'}$ and $\tilde{h} = \tilde{h}_{i'}$ defined above, the $\mathbb{F}'$-linear functions $\bar{L}\colon \mathbb{F}'^{f/f'} \to \mathbb{F}'$ are in bijective correspondence with the R'-linear functions $L\colon R/\tilde{\mathfrak{p}} \to R'/\mathfrak{p}'$.*

Proof. For any $\mathbb{F}'$-linear function $\bar{L}$, we claim that $L = h'^{-1} \circ \bar{L} \circ \tilde{h}$ is the corresponding R'-linear function. To see this, note that by Lemma 2.3 and the fact that $\tilde{h}$ is a ring homomorphism, for any $r' \in R'$ and $a \in R/\tilde{\mathfrak{p}}$ we have

$$\tilde{h}(r' \cdot a) = \tilde{h}(r' \bmod \tilde{\mathfrak{p}}) \odot \tilde{h}(a) = h'(r' \bmod \mathfrak{p}') \cdot \tilde{h}(a) \in \mathbb{F}'^{f/f'},$$

where multiplication $\odot$ in $\mathbb{F}'^{f'}$ and $\mathbb{F}^f$ is coordinate-wise. By $\mathbb{F}'$-linearity of $\bar{L}$ and the fact that h' is a ring homomorphism, we have

$$L(r' \cdot a) = h'^{-1}(\bar{L}(\tilde{h}(r' \cdot a))) = h'^{-1}\big(\, h'(r' \bmod \mathfrak{p}') \cdot \bar{L}(\tilde{h}(a))\,\big) = r' \cdot L(a) \in R'/\mathfrak{p}',$$

as desired. The other direction proceeds essentially identically, with $\bar{L} = h' \circ L \circ \tilde{h}^{-1}$. □

An application of the Chinese Remainder Theorem with the prime ideals $\tilde{\mathfrak{p}}_{i'}$ in R, combined with Lemma 2.4, immediately yields the following corollary.

Corollary 2.5. *Let $\mathfrak{p}' = \prod_{i' \in G'} \mathfrak{p}'_{i'}$ and $\mathfrak{p} = \prod_{i' \in G'} \tilde{\mathfrak{p}}_i$ be the radicals of pR' and pR, respectively. Then under the isomorphisms $\{h_{i'}\}_{i' \in G'}$ and $\{\tilde{h}_{i'}\}_{i' \in G'}$ defined above, the R'-linear functions $L\colon R/\mathfrak{p} \to R'/\mathfrak{p}'$ are in bijective correspondence with the functions $\bar{L}\colon (\mathbb{F}'^{f/f'})^{f'} \to \mathbb{F}'^{f'}$ of the form*

$$\bar{L}(*)(\vec{a}_{i'})_{i' \in G'} = \big(\bar{L}_{i'}(\vec{a}_{i'})\big)_{i' \in G'},$$

where every $\bar{L}_{i'}\colon \mathbb{F}'^{f/f'} \to \mathbb{F}'$ is $\mathbb{F}'$-linear.

We note that given a function $\bar{L}\colon (\mathbb{F}'^{f/f'})^{f'} \to \mathbb{F}'^{f'}$ as in Corollary 2.5, we can efficiently find an R'-linear function $\hat{L}\colon R \to R'$ that induces the corresponding L: first, fix an arbitrary R'-basis $B = \{b_j\}$ of R. Then, using the isomorphisms $h'_{i'}$ and $\tilde{h}_{i'}$, the values of $L(b_j \bmod \mathfrak{p}) \in R'/\mathfrak{p}'$ are determined by $\bar{L}$, and uniquely define L by R'-linearity. We can then define each $\hat{L}(b_j) \in R'$ to be an arbitrary representative of $L(b_j \bmod \mathfrak{p})$; these choices uniquely determine $\hat{L}$, by R'-linearity. Finally, we can represent $\hat{L}$ explicitly in trace form as $\hat{L}(a) = \mathrm{Tr}_{K/K'}(r \cdot a)$ for

some $r \in K$: recalling that K is a vector space over K' with K'-basis B, we have a full-rank system of linear equations $\hat{L}(b_j) = \mathrm{Tr}_{K/K'}(r \cdot b_j) \in K'$, which we can solve to obtain $r \in K$.

Looking ahead, in our application to homomorphic computation we will have certain linear functions that we want to evaluate (e.g., projection functions), and we will do so by finding the corresponding constant r, then multiplying by r and taking the trace (see Section 3.3 for further details). To apply these steps in the context of a homomorphic encryption scheme, we need the notion of the *dual* of the ring of integers, described next.

2.1.4. Duality

An important and useful object in K is the *dual* of R (also known as the *codifferent* of K), defined as

$$R^\vee = \{a \in K : \mathrm{Tr}_{K/\mathbb{Q}}(aR) \subseteq \mathbb{Z}\} \supseteq R.$$

Because $\mathrm{Tr}_{K/\mathbb{Q}} = \mathrm{Tr}_{K'/\mathbb{Q}} \circ \mathrm{Tr}_{K/K'}$, it is easy to verify that also $R^\vee = \{a \in K : \mathrm{Tr}_{K/K'}(aR) \subseteq R'^\vee\}$. Therefore, we have the convenient equation

$$\mathrm{Tr}_{K/K'}(R^\vee) = R'^\vee. \tag{3}$$

Note that by contrast, frequently $\mathrm{Tr}_{K/K'}(R)$ does *not* equal R', but is instead some proper ideal of it.[5] Many other algebraic and geometric advantages of working with $R^\vee$ instead of R are discussed in [15,16].

The codifferent is a principal fractional ideal, i.e., $R^\vee = t^{-1}R$ for some $t \in R$ (which is not unique). Therefore, division by t induces a bijection from R to $R^\vee$, and from any quotient ring $R_\mathfrak{p} = R/\mathfrak{p}$ to $R_\mathfrak{p}^\vee = R^\vee/\mathfrak{p}R^\vee$. Although the target objects are *not* rings (because $R^\vee \cdot R^\vee \not\subseteq R^\vee$), they are R-modules, and the bijections are R-module isomorphisms.

Of course, we also have $R'^\vee = t'^{-1}R'$ for some $t' \in R'$. By Equation (3) and K'-linearity of the trace, for any ideal $\mathfrak{p}$ in R', we have

$$\mathrm{Tr}_{K/K'}(R_\mathfrak{p}^\vee) = \mathrm{Tr}_{K/K'}(R^\vee/\mathfrak{p}R^\vee) = R'^\vee/\mathfrak{p}R'^\vee = R_\mathfrak{p}'^\vee.$$

In the previous subsection we considered R'-linear functions $L\colon R \to R'$ (or their induced functions $R_p \to R'_p$), which can always be expressed as $L(a) = \mathrm{Tr}_{K/K'}(r^\vee \cdot a)$ for some fixed $r^\vee \in K$. Typically, $r^\vee$ is *not* in R because $\mathrm{Tr}_{K/K'}(R) \neq R'$, but it is easy to see that $r^\vee \in t'R^\vee$ always, because if not, then $\mathrm{Tr}_{K/K'}(r^\vee R) \not\subseteq t'R'^\vee = R'$. For the purposes of our field-switching procedure, it will be more convenient to instead work with corresponding R'-linear functions from $R^\vee$ to $R'^\vee$, which can be represented in trace form by elements of R. Namely, for an R'-linear function $L : R \to R'$, where $L(a) = \mathrm{Tr}_{K/K'}(r^\vee \cdot a)$ for some $r^\vee \in t'R^\vee$, we will consider the corresponding function

$$L^\vee\colon R^\vee \to R'^\vee, \quad L^\vee(a^\vee) = L(t \cdot a^\vee)/t' = \mathrm{Tr}_{K/K'}((t/t')r^\vee \cdot a^\vee) = \mathrm{Tr}_{K/K'}(r \cdot a^\vee),$$

[5] This is easily seen, e.g., for $R = \mathbb{Z}[\zeta_{2^k}]$ and $R' = \mathbb{Z}$, where $\mathrm{Tr}(R) = 2^{k-1}R'$ because $\mathrm{Tr}(1) = 2^{k-1}$ and $\mathrm{Tr}(\zeta_{2^k}^j) = 0$ for $j = 1, \ldots, 2^{k-1} - 1$.

which is represented by $r = (t/t')r^\vee \in R$.

Following [16], we extend the operation $[\cdot]_q$ to $R_p^\vee$ by fixing a particular $\mathbb{Z}$-basis of $R^\vee$ (and $\mathbb{Z}_q$-basis of $R_q^\vee$), called the *decoding basis*, and representing the argument as a $\mathbb{Z}_q$-combination of the basis vectors and applying the $[\cdot]_q$ operation to each of its coefficients. It is shown in [16, Section 6.2] that every sufficiently short (as always, under the canonical embedding) $e \in R^\vee$ is indeed the "canonical" representative of its coset modulo $qR^\vee$. Specifically, if $\|e\| < q/(2\sqrt{n})$ then $[e \bmod qR^\vee]_q = e$.

2.1.5. Good Bases of R and $R^\vee$

We now have almost all the ingredients we need to describe the homomorphic cryptosystem and our field-switching transformation. The final background material we need concerns the geometry of R as a module over R' (respectively, $R^\vee$ as a module over $R'^\vee$). Specifically, we construct certain "good" bases of the ring R and its dual $R^\vee$ in terms of R' and $R'^\vee$ (respectively), and prove some of their useful geometrical properties. This (somewhat technical) material is used only in Section 3.1, where we prove the hardness of ring-LWE over K with secret in R', assuming its hardness over K' with secret in R'.

Since K is a vector space of dimension n/n' over K', the field K has a K'-*basis* (which is not unique), i.e., a set of n/n' elements of K that are linearly independent over K', so that every element of K can be represented uniquely as a K'-linear combination of the basis elements. Similarly, an R'-*basis* of R is a set of n/n' elements in R, such that every element of R can be represented uniquely as an R'-linear combination of the basis elements. An $R'^\vee$-basis of $R^\vee$ is defined analogously.

We wish to construct an R'-basis of R, and a corresponding dual $R'^\vee$-basis of $R^\vee$ (any of which are K'-bases of K), which are "good" in the following sense: for any vector of K'-coefficients (with respect to the basis) which are *short* under σ', the corresponding K-element is also short under σ. More formally, represent an ordered K'-basis of K as a vector $\vec{b} = (b_j) \in K^{n/n'}$, and similarly for an arbitrary vector of K'-coefficients $\vec{a} = (a_j) \in K'^{(n/n')}$, which defines the K-element $a = \langle \vec{a}, \vec{b} \rangle = \sum_j a_j \cdot b_j$. Then by linearity, the basis $\vec{b}$ induces a matrix $B \in \mathbb{C}^{n \times n}$ such that

$$\sigma(a) = B \cdot \sigma'(\vec{a}), \quad \text{where } \sigma'(\vec{a}) = \big(\sigma'(a_j)\big)_j. \tag{4}$$

We seek an R'-basis $\vec{b}$ of R for which B (nearly) preserves Euclidean norms up to some scaling factor, i.e., all of its singular values are (nearly) equal.

In addition, for any K'-basis $\vec{b} = (b_j)$ of K, its *dual* K'-basis $\vec{b}^\vee = (b_j^\vee) \subseteq K$ is uniquely defined by the linear constraints $\mathrm{Tr}_{K/K'}(b_j \cdot b_{j'}^\vee) = 1$ if $j = j'$, and 0 otherwise. It is a straightforward exercise to verify that if $\vec{b}$ is an R'-basis of R, then $\vec{b}^\vee$ is an $R'^\vee$-basis of $R^\vee$. Moreover, the matrix $B^\vee$ induced by $\vec{b}^\vee$ is $B^\vee = B^{-T}$, so its singular values are simply the inverses of those of B.

Lemma 2.6. *Let $\hat{m} = m/2$ if m is even and m' is odd, otherwise $\hat{m} = m$, and let $r = \mathrm{rad}(m)/\mathrm{rad}(m')$ be the product of all primes that divide m but not m'. There*

exists an efficiently computable R'-basis $\vec{b}$ of R, for which the corresponding matrix B has largest and smallest singular values

$$s_1(B) = \sqrt{\hat{m}/m'} \quad \textit{and} \quad s_n(B) = \sqrt{m/(rm')},$$

respectively. In particular, if $r \in \{1, 2\}$ then B is a unitary matrix scaled by a $\sqrt{\hat{m}/m'}$ factor.

Lemma 2.6 implies that for any $\vec{a} \in K'^{(n/n')}$ defining $a = \langle \vec{a}, \vec{b} \rangle \in K$ and $a^\vee = \langle \vec{a}, \vec{b}^\vee \rangle \in K$,

$$\|\sigma(a)\| \leq \sqrt{\hat{m}/m'} \cdot \|\sigma'(\vec{a})\| \quad \text{and} \quad \|\sigma(a^\vee)\| \leq \sqrt{rm'/m} \cdot \|\sigma'(\vec{a})\|. \tag{5}$$

More generally, if the a_j are independent and have Gaussian distributions over (the canonical embedding of) K', then a and $a^\vee$ also have (possibly non-spherical) Gaussian distributions over K.[6] Since we are not too concerned with the exact distributions, we omit a precise calculation, which is standard. However, one particular case of interest is when the a_j are all i.i.d. according to a spherical Gaussian of parameter s, and $r \in \{1, 2\}$ so that B (respectively, $B^\vee$) is a scaled unitary matrix. Then because spherical Gaussians are invariant under unitary transformations, a (resp., $a^\vee$) is distributed according to a spherical Gaussian of parameter $s\sqrt{\hat{m}/m'}$ (resp., $s\sqrt{m'/\hat{m}}$).

The remainder of this subsection is devoted to proving Lemma 2.6. We denote the k-dimensional identity matrix by I_k, we use $\otimes$ to denote the Kronecker (or tensor) product of vectors and matrices, and we apply functions to vectors and matrices component-wise.

Following the treatment given in [16], let $m = \prod_\ell m_\ell$ be the prime-power factorization of m, i.e., the $m_\ell > 1$ are powers of distinct primes. The ring $R = \mathbb{Z}[\zeta_m]$ has the following $\mathbb{Z}$-basis $\vec{p}$, which is called the "powerful" basis:

$$\vec{p} = \bigotimes\nolimits_\ell \vec{p}_{m_\ell}\ , \quad \text{where } \vec{p}_{m_\ell} = (*)\zeta_{m_\ell}^{j}{}_{j \in [*]\varphi(m_\ell)}.$$

The set $\vec{p}_{m_\ell}$ is called the "power" $\mathbb{Z}$-basis of $\mathbb{Z}[\zeta_{m_\ell}] = \mathbb{Z}[\zeta_m^{m/m_\ell}] \subseteq R$.

Similarly, let $m' = \prod_\ell m'_\ell$ where each m'_ℓ divides m_ℓ, i.e., they are both powers of the same prime (though possibly $m'_\ell = 1$). Then the powerful $\mathbb{Z}$-basis of R' is defined as $\vec{p}' = \bigotimes_\ell \vec{p}_{m'_\ell}$, where the power bases $\vec{p}_{m'_\ell}$ are defined as above. Notice that when $m'_\ell > 1$, there is a bijective correspondence between $j \in [\varphi(m_\ell)]$ and $(j', k) \in [\varphi(m'_\ell)] \times [m_\ell/m'_\ell]$, via $j = (m_\ell/m'_\ell)j' + k$. Therefore, the power bases $\vec{p}_{m_\ell}$ factor as

$$\vec{p}_{m_\ell} = \vec{p}_{m'_\ell} \otimes \vec{b}_\ell, \quad \text{where } \vec{b}_\ell = \begin{cases} ([])\zeta_{m_\ell}^{k}{}_{k \in [m_\ell/m'_\ell]} & \text{if } m'_\ell > 1 \\ \vec{p}_{m_\ell} & \text{if } m'_\ell = 1. \end{cases}$$

Hence, using the commutativity of the Kronecker product (up to some permutation) we can factor the powerful basis $\vec{p}$ of R as

[6] To be completely formal, the Gaussians should be over continuous spaces of the form $K \otimes_{\mathbb{Q}} \mathbb{R}$; see [16].

$$\vec{p} = \vec{p}' \otimes \vec{b}, \quad \text{where } \vec{b} = \bigotimes_\ell \vec{b}_\ell. \tag{6}$$

Because $\vec{p}'$ is a $\mathbb{Z}$-basis of R', it follows that $\vec{b}$ is an R'-basis of R. We next calculate the matrix $B \in \mathbb{C}^{n \times n}$ induced by $\vec{b}$, and verify that it indeed satisfies the claims in the lemma statement.

Following [16, Section 3], for any prime power $\tilde{m}$ we define $\mathrm{CRT}_{\tilde{m}}$ to be the complex $\varphi(\tilde{m})$-by-$\varphi(\tilde{m})$ matrix with $\omega_{\tilde{m}}^{i \cdot j}$ in its ith row and jth column, for $i \in \mathbb{Z}_{\tilde{m}}^*$ and $j \in [\varphi(\tilde{m})]$. Using the prime-power factorizations of our m, m', we define $\mathrm{CRT}_m = \bigotimes_\ell \mathrm{CRT}_{m_\ell}$ and $\mathrm{CRT}_{m'} = \bigotimes_\ell \mathrm{CRT}_{m'_\ell}$. Then up to a permutation of the rows (determined by the CRT correspondence between $\mathbb{Z}_m^*$ and $\prod_\ell \mathbb{Z}_{m_\ell}^*$), we have

$$\sigma(\vec{p}^T) = \mathrm{CRT}_m,$$

i.e., the columns of CRT_m are $\sigma(p_j)$ for each entry p_j of the row vector $\vec{p}^T$. In particular, $\sigma(\langle \vec{c}, \vec{p} \rangle) = \mathrm{CRT}_m \cdot \vec{c}$ for any $\vec{c} \in \mathbb{Q}^n$. Similarly, $\sigma'((\vec{p}')^T) = \mathrm{CRT}_{m'}$ up to a row permutation.

We now claim that, up to some permutations of B's rows and columns,

$$B = \mathrm{CRT}_m \cdot (*) \mathrm{CRT}_{m'}^{-1} \otimes I_{n/n'} = \bigotimes_\ell \big([] \big] \mathrm{CRT}_{m_\ell} \cdot ([] \big] \mathrm{CRT}_{m'_\ell}^{-1} \otimes I_{\varphi(m_\ell)/\varphi(m'_\ell)}, \tag{7}$$

where the second equality follows by the mixed-product property and the commutativity (up to row and column permutations) of the Kronecker product. To see the first equality, notice that for any $\vec{a} \in K'^{(n/n')}$ defining $a = \langle \vec{a}, \vec{b} \rangle \in K$, the matrix $(\mathrm{CRT}_{m'}^{-1} \otimes I)$ maps from (a suitable permutation of) the concatenated embeddings $\sigma'(\vec{a})$, to a vector $\vec{c} \in \mathbb{Z}^n$ of coefficients such that $\vec{a} = \langle \vec{c}, \vec{p}' \otimes I_{n/n'} \rangle$. In addition,

$$a = \langle \vec{a}, \vec{b} \rangle = \vec{c}^T \cdot (\vec{p}' \otimes I_{n/n'}) \cdot \vec{b} = \langle \vec{c}, \vec{p}' \otimes \vec{b} \rangle = \langle \vec{c}, \vec{p} \rangle.$$

Therefore, $\sigma(a) = \mathrm{CRT}_m \cdot \vec{c} = \mathrm{CRT}_m \cdot (\mathrm{CRT}_{m'}^{-1} \otimes I) \cdot \sigma'(\vec{a})$, as desired.

Now, by the last expression in Equation (7), and because singular values are multiplicative under the Kronecker product, from now on we drop all the ℓ subscripts, and assume without loss of generality that m and m' are powers of the same prime p (where possibly $m' = 1$). We analyze the singular values of $\mathrm{CRT}_m(\mathrm{CRT}_{m'}^{-1} \otimes I)$, for the cases $m' = 1$ and $m' > 1$. In the first case, clearly $\mathrm{CRT}_{m'} = I_1$, and it is shown in [16, Section 4] that the largest singular value of CRT_m is $\sqrt{m/2}$ if m is even and $\sqrt{m}$ otherwise, and its smallest singular value is $\sqrt{m/p}$.

For the case $m' > 1$, it follows from the decompositions given in [16, Section 3] that, up to some row permutation,

$$\mathrm{CRT}_m = \sqrt{m/p} \cdot Q \cdot (\mathrm{CRT}_p \otimes I_{m/p})$$

for some unitary matrix Q, and similarly for $\mathrm{CRT}_{m'}$. Then a routine calculation using elementary properties of the Kronecker product reveals that $\mathrm{CRT}_m(\mathrm{CRT}_{m'}^{-1} \otimes I)$ is some unitary matrix scaled by a $\sqrt{m/m'}$ factor, so all its singular values are $\sqrt{m/m'}$. This completes the proof of Lemma 2.6.

2.2. Homomorphic Cryptosystems

In ring-LWE-based cryptosystems for arbitrary cyclotomics [16] (generalizing those of [15,4,3]), the plaintext space is R_p for some integer $p \geq 2$ that is coprime with all the odd primes dividing m. We assume that p is prime, which is without loss of generality by the Chinese Remainder Theorem. Ciphertexts are elements of $(R_q^\vee)^2$ for some integer q that is coprime with p, and the secret key is some $s \in R$. A ciphertext $c = (c_0, c_1) \in (R_q^\vee)^2$ that encrypts a plaintext $b \in R_p$ with respect to s satisfies the decryption relation

$$c_0 + c_1 \cdot s = e \pmod{qR^\vee} \tag{8}$$

for some sufficiently short $e \in R^\vee$ such that $t \cdot e = b \pmod{*}pR$. (Recall that $R^\vee = t^{-1}R$ for some $t \in R$, so $t \cdot e \in R$.) We refer to e as the *noise* of the ciphertext. Throughout this work we implicitly assume that the modulus q is large enough relative to $\|e\|$, so that $[c_0 + c_1 \cdot s]_q = e \in R^\vee$ (see Section 2.1.4 above). Therefore, the decryption algorithm can simply compute e and output $t \cdot e \bmod pR$. As shown in [4,3,16], this system (augmented by some additional public values, for greater efficiency) supports additive and multiplicative homomorphisms.

3. The Field-Switching Procedure

Our procedure performs the following operation. Given a big-field ciphertext $c \in (R_q^\vee)^2$ that encrypts a plaintext $b \in R_p$ with respect to a big-ring secret key $s \in R$, and a description of an R'-linear function $L\colon R_\mathfrak{p} \to R'_{\mathfrak{p}'}$ to apply to the plaintext (where recall that $\mathfrak{p}$ and $\mathfrak{p}'$ are the radicals of p in R and R', respectively), it outputs a small-field ciphertext $c' \in (R_q'^\vee)^2$ that encrypts $b' = L(b) \in R'_{\mathfrak{p}'}$ with respect to some small-ring secret key $s' \in R'$. (Recall that Corollary 2.5 characterizes how L corresponds to the induced function $\bar{L}\colon \mathbb{F}^f \to \mathbb{F}'^{f'}$ that is applied to the vector of finite field elements encoded by b.)

The procedure consists of the following three steps:

1. **Switch to a small-ring secret key.** We use the key-switching method from [5, 3,16] to produce a ciphertext which is still over the big field K and encrypts the same plaintext $b \in R_\mathfrak{p}$, but with respect to a secret key $s' \in R' \subseteq R$ belonging to the small subring.
2. **Multiply by an appropriate (short) scalar.** We multiply the components of the resulting ciphertext by a short element $r \in R$ that corresponds to the desired R'-linear function to be applied to the input plaintext b.
3. **Map to the small field.** We map the resulting big-field ciphertext (over $R_q^\vee$) to a small-field ciphertext (over $R_q'^\vee$) simply by taking the trace $\mathrm{Tr}_{K/K'}$ of its two components. The resulting ciphertext will still be with respect to the small-ring secret key $s' \in R'$, but will encrypt the plaintext $b' = L(b) \in R'_{\mathfrak{p}'}$.

Note that Steps 2 and 3 can be repeated multiple times on the same ciphertext (from Step 1), to apply several different R'-linear functions. In this way, the entire input plaintext can be preserved, but in a decomposed form.

3.1. Step 1: Switching to a Small-Ring Secret Key

To switch to a small-field secret key, we publish a "key-switching hint," which essentially encrypts the big-ring secret key $s \in R$ under the small-ring key $s' \in R'$, using ciphertexts over the big field. Note that encrypting s under a small-ring secret key s' has security implications, since the dimension of the underlying RLWE problem is smaller. In our case, though, the ultimate goal is to switch to a ciphertext over the smaller field, so we will not lose any additional security by publishing the hint. Indeed, we show below that assuming the hardness of the decision RLWE problem in the small field, the key-switching hint reveals nothing about the big-ring secret key. The essence of that claim is Lemma 3.1 below, which says (informally) that RLWE in the big field, with secret chosen in the small ring $R' \subseteq R$, is no easier than RLWE in the small field.

Ring-LWE. The ring-LWE (RLWE) problem [15] (in K) with *continuous* error is parameterized by a modulus q, a "secret distribution" υ over R, and an "error distribution" ψ over K, which is usually a Gaussian (in the canonical embedding) and is therefore concentrated on short elements.[7] For $s \in R$, define the distribution $A_{s,\psi}$ that is sampled by choosing $\alpha \in R_q^\vee$ uniformly at random, choosing $\epsilon \leftarrow \psi$, and outputting the pair $(\alpha, \beta = \alpha \cdot s + \epsilon \bmod qR^\vee) \in R_q^\vee \times K/qR^\vee$. One equivalent form of the (average-case) decision $\mathrm{RLWE}_{q,\psi,\upsilon}$ problem (in K) is, given some ℓ pairs $(\alpha_i, \beta_i) \in R_q^\vee \times K/qR^\vee$, distinguish between the following two cases: in one case, the pairs are chosen independently from $A_{s,\psi}$ for a random $s \leftarrow \upsilon$ (which remains the same for all samples); in the other case, the pairs are all independent and uniformly random over $R_q^\vee \times K/qR^\vee$. For appropriate parameters q, ψ, υ and ℓ, solving this decision problem with non-negligible distinguishing advantage is as hard as approximating the shortest vector problem on ideal lattices in R, via a quantum reduction. See [15,16] for precise statements and further details.

Let $\vec{b}^\vee = (b_j^\vee)_{j \in [n/n']}$ be any $R'^\vee$-basis of $R^\vee$, and hence a K'-basis of K. Then for any error distribution ψ' over K', we can define an error distribution ψ over K as $\psi = \langle \psi'^{(n/n')}, \vec{b}^\vee \rangle$, i.e., a sample from ψ is generated by choosing independent $\epsilon_j \leftarrow \psi'$ (for $j \in [n/n']$) and outputting $\epsilon = \sum_j \epsilon_j b_j^\vee \in K$.

Lemma 3.1. *Let ψ' be an error distribution over K', and let $\psi = \langle \psi'^{(n/n')}, \vec{b}^\vee \rangle$ be the error distribution over K as described above. If the decision $\mathrm{RLWE}_{q,\psi',\upsilon'}$ problem (in K') is hard for some distribution υ' over $R' \subseteq R$, then the decision $\mathrm{RLWE}_{q,\psi,\upsilon'}$ problem (in K) is also hard.*

Although the lemma holds for any $R'^\vee$-basis of $R^\vee$, it is most useful with a basis having "good geometric properties." Specifically, in our case we need the property that if ψ' is concentrated on short elements of K', then ψ is similarly concentrated on short elements of K. Such a basis $\vec{b}^\vee$ is constructed in Lemma 2.6 of Section 2.1.5. For example, if ψ' is a continuous (spherical) Gaussian with parameter s and $r = \mathrm{rad}(m)/\mathrm{rad}(m') = 1$, then ψ' is a spherical Gaussian with parameter $s\sqrt{m'/m} = s\sqrt{n'/n}$.[8]

[7] Again, to be completely formal, a Gaussian should be defined over $K_{\mathbb{R}}$; see Footnote 6.

[8] Note that the factor $\sqrt{n'/n} \leq 1$ does not really amount to any effective decrease in the noise, because the "sparsity" of $R'^\vee$ versus $R^\vee$ is greater by a corresponding factor.

Proof. It suffices to give an efficient, deterministic reduction that takes n/n' pairs $(\alpha_j, \beta_j) \in R_q'^\vee \times K'/qR'^\vee$ and outputs a single pair $(\alpha, \beta) \in R^\vee \times K/qR^\vee$, with the following properties: if the pairs (α_j, β_j) are i.i.d. according to $A_{s',\psi'}$ for some $s' \in R'$, then (α, β) is distributed according to $A_{s,\psi}$; and if the pairs (α_j, β_j) are independent and uniformly random, then (α, β) is uniformly random. The reduction simply outputs $(\alpha = \langle \vec{\alpha}, \vec{b}^\vee \rangle, \beta = \langle \vec{\beta}, \vec{b}^\vee \rangle)$, where $\vec{\alpha} = (\alpha_j)_j$ and $\vec{\beta} = (\beta_j)_j$.

Since $\vec{b}^\vee$ is an $R'^\vee$-basis of $R^\vee$ and hence an $R_q'^\vee$-basis of $R_q^\vee$, it is immediate that the reduction maps the uniform distribution to the uniform distribution. On the other hand, if the samples $(\alpha_j \beta_j)$ are drawn from $A_{s',\psi'}$, i.e, $\beta_j = \alpha_j \cdot s' + \epsilon_j \bmod qR'^\vee$ for $\epsilon_j \leftarrow \psi$, then α is still uniformly random, and

$$\beta = \langle \vec{\beta}, \vec{b}^\vee \rangle = \langle \vec{\alpha}, \vec{b}^\vee \rangle \cdot s' + \langle \vec{\epsilon}, \vec{b}^\vee \rangle = \alpha \cdot s' + \epsilon \pmod{qR^\vee},$$

where $\vec{\epsilon} = (\epsilon_j)_j$ and ϵ has distribution ψ. This completes the proof. □

Key switching. In [5,3,16] it is shown how, given an $s \in R$ and sufficiently many RLWE samples (over K) with short noise and any secret $s' \in R$, it is possible to generate a "key-switching hint" with the following functionality: given the hint and any valid ciphertext c (over K) encrypted under s and with sufficiently short noise, it is possible to efficiently generate a ciphertext c' (also over K) with short noise encrypted under s'. Moreover, the hint is indistinguishable from uniformly random over its domain (even given s), assuming that the RLWE samples are.

For our transformation, we apply Lemma 3.1 using the "good basis" $\vec{b}^\vee$ from Lemma 2.6, thus obtaining RLWE samples over K relative to the secret $s' \in R' \subseteq R$, with noise distribution ψ which is concentrated on short vectors, and with security based on the hardness of $\mathrm{RLWE}_{q,\psi',\upsilon'}$ problem in K'. We then construct the key-switching hint from these samples as described in [16, Section 8.3],

3.2. Steps 2 and 3: Mapping to the Small Field

Our goal now is to transform a valid big-field ciphertext $c = (c_0, c_1) \in (R_q^\vee)^2$, which encrypts some $b \in R_\mathfrak{p}$ with respect to some secret key $s' \in R' \subseteq R$, into a small-field ciphertext $c' = (c_0', c_1') \in (R_q'^\vee)^2$ that encrypts the related plaintext $b' = L(b)$ with respect to the same secret key s', where $L \colon R_\mathfrak{p} \to R'_{\mathfrak{p}'}$ is any desired R'-linear function.

The process works as follows:

1. Since L is R'-linear, by the discussion at the end of Section 2.1.3 and in Section 2.1.4, we can find some $r^\vee \in t'R^\vee$ such that $L(a) = \mathrm{Tr}_{K/K'}(r^\vee \cdot a) \bmod \mathfrak{p}'$.
2. We then find a *short* representative $r \in (t/t')r^\vee + pR \in R_p$, using a "good" basis of pR (i.e., one that has small singular values under σ, e.g., the "powerful" basis as constructed in Section 2.1.5).
   The chosen r defines the R'-linear function $L^\vee \colon R^\vee \to R'^\vee$ of the form $L^\vee(a^\vee) = \mathrm{Tr}_{K/K'}(r \cdot a^\vee)$, whose induced function from $R_\mathfrak{p}^\vee$ to $R_{\mathfrak{p}'}^\vee$ satisfies

$$t' \cdot L^\vee(a^\vee) = L(t \cdot a^\vee) \pmod{\mathfrak{p}'}. \tag{9}$$

3. We obtain our small-field ciphertext by applying $L^\vee$ (or more precisely, the induced function from $R_q^\vee$ to $R_q'^\vee$) to c_0, c_1, setting

$$c_i' = L^\vee(c_i) = \mathrm{Tr}_{K/K'}(r \cdot c_i) \in R_q'^\vee, \quad i = 0, 1.$$

Lemma 3.2. *The ciphertext $c' = (c_0', c_1')$ is an encryption of $b' = L(b) \in R'_{\mathfrak{p}'}$ under secret key $s' \in R'$, with noise $e' = L^\vee(e) \in R'^\vee$ of length $\|e'\| \le \|e\| \cdot \|r\|_\infty \cdot \sqrt{n/n'}$, where e is the noise in the original ciphertext c.*

We note that the factor $\sqrt{n/n'}$ in the bound on $\|e'\|$ does not actually amount to any effective increase in the noise, because the dimension has decreased by a corresponding factor, and hence the size of e' relative to $R'^\vee$ remains the same as that of e relative to $R^\vee$. More precisely, the original ciphertext c decrypts correctly if $q > 2\sqrt{n}\|e\|$, whereas c' decrypts correctly if $q > 2\sqrt{n'}\|e'\|$ (see Section 2.1.4). Therefore, the only practical increase in the noise is due solely to $\|r\|_\infty$.

Proof. We need to show three things: that $\|e'\|$ is bounded as claimed, that $c_0' + c_1' \cdot s = e' \pmod{*} qR'^\vee$, and that $t' \cdot e' = b' = L(b) \pmod{*}\mathfrak{p}'$.

1. The first claim follows immediately by Corollary 2.2 and the inequality $\|r \cdot e\| \le \|r\|_\infty \cdot \|e\|$.
2. For the second claim, recall that $c_0 + c_1 \cdot s = e \pmod{*} qR^\vee$. Then because the induced function $L^\vee \colon R_q^\vee \to R_q'^\vee$ is R'-linear and $s' \in R'$, we have

$$c_0' + c_1' \cdot s' = L^\vee(c_0 + c_1 \cdot s') = L^\vee(e) = e' \pmod{R_q'^\vee}.$$

3. For the last claim, because $t \cdot e = b \bmod pR$ and by Equation (9), we have

$$t' \cdot e' = t' \cdot L^\vee(e) = L(t \cdot e) = L(b) \pmod{\mathfrak{p}'}. \qquad \square$$

3.3. Applying the Field-Switching Procedure

A typical application of the field-switching procedure during homomorphic evaluation of some circuit will begin with a big-field ciphertext that encrypts an array of plaintext values in the subfield $\mathbb{F}'$, as embedded in $\mathbb{F}$.[9] The above procedure is then applied to decompose the ciphertext into a number of small-field ciphertexts, each encrypting a subset of the plaintext values. Since big-field ciphertexts have room for f plaintext elements, but small-field ciphertexts can only hold f' elements, we may need up to f/f' small-field ciphertexts to hold all the plaintext values that we are interested in. That is, we apply our field-switching transformation using the f'-fold concatenations $\bar{L}_i^{f'}$ of the $\mathbb{F}'$-linear selection functions $\bar{L}_i \colon \mathbb{F}^{f/f'} \to \mathbb{F}'$, $i \in [f/f']$, where $\bar{L}_i$ just selects the ith value (in $\mathbb{F}'$).[10]

[9] For example, when evaluating AES homomorphically, we would have plaintext values from $\mathbb{F}_{2^8}$ even though $\mathbb{F}$ may be a larger field such as $\mathbb{F}_{2^{16}}$ or $\mathbb{F}_{2^{24}}$, etc.

[10] More precisely, $\bar{L}_i(\vec{a}) = \mathrm{Tr}_{\mathbb{F}/\mathbb{F}'}(\rho \cdot a_i)$ for some $\rho \in \mathbb{F}$ such that $\mathrm{Tr}_{\mathbb{F}/\mathbb{F}'}(\rho) = 1$, so that $\bar{L}_i(\vec{a}) = a_i$ for any $a_i \in \mathbb{F}'$, by $\mathbb{F}'$-linearity.

Referring to Figure 1 for an example, the big-field ciphertext holds (up to) six plaintext values, and each small-field ciphertext can hold two values, with the big-field plaintext "slots" corresponding to $\mathfrak{p}_1, \mathfrak{p}_{15}, \mathfrak{p}_{22}$ lying over the small-field plaintext slot of $\mathfrak{p}'_1$, and the big-field slots corresponding to $\mathfrak{p}_3, \mathfrak{p}_{17}, \mathfrak{p}_{31}$ lying over the small-field plaintext slot of $\mathfrak{p}'_3$. Then we can produce three small-field ciphertexts, using the three selection functions

$$\begin{aligned}(x_1, x_{15}, x_{22},\ x_3, x_{17}, x_{31}) &\mapsto (\ x_1\ ,\ x_3\ ),\\ (x_1, x_{15}, x_{22},\ x_3, x_{17}, x_{31}) &\mapsto (x_{15}\ , x_{17}),\\ (x_1, x_{15}, x_{22},\ x_3, x_{17}, x_{31}) &\mapsto (x_{22}\ , x_{31}).\end{aligned}$$

Acknowledgments

The first and second authors are supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center (DoI/NBC) contract number D11PC20202. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

The third author is supported by the National Science Foundation under CAREER Award CCF-1054495, by the Alfred P. Sloan Foundation, and by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under Contract No. FA8750-11-C-0098. The views expressed are those of the authors and do not necessarily reflect the official policy or position of the National Science Foundation, the Sloan Foundation, DARPA or the U.S. Government.

The fourth author is supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant COED–EP/I03126X, and by a Royal Society Wolfson Merit Award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the European Commission or EPSRC.

References

[1] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *CRYPTO'09*, volume 5677 of *Lecture Notes in Computer Science*, pages 595–618. Springer, 2009.

[2] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *CRYPTO'12*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012. Available at `http://eprint.iacr.org/2012/078`.

[3] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science (ITCS'12)*, ACM, 2012. Available at `http://eprint.iacr.org/2011/277`.

[4] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[5] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS'11*. IEEE Computer Society, 2011.

[6] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO'12*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012. Available at `http://eprint.iacr.org/2011/535`.

[7] Leo Ducas and Alain Durmus. Ring-LWE in Polynomial Rings. In *PKC'12*, volume 7293 of *Lecture Notes in Computer Science*, pages 34–51. Springer, 2012. Available at `http://eprint.iacr.org/2012/235`

[8] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In *EUROCRYPT'08*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 2008.

[9] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC'09*, pages 169–178. ACM, 2009.

[10] Craig Gentry, Shai Halevi, Chris Peikert and Nigel P. Smart. Ring Switching in BGV-Style Homomorphic Encryption. In *SCN'12*, volume 7485 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2012. Available at `http://eprint.iacr.org/2012/240`.

[11] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT'12*, volume 7237 of *Lecture Notes in Computer Science*, pages 446-464, 2012. Available at `http://eprint.iacr.org/2011/566`.

[12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping for fully homomorphic encryption. In *PKC'12*, volume 7293 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012. Available at `http://eprint.iacr.org/2011/680`.

[13] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO'12*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012. Available at `http://eprint.iacr.org/2012/099`.

[14] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption In *STOC'12 Proceedings of the 44th symposium on Theory of Computing*, Pages 1219-1234. ACM, 2012.

[15] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Journal of the ACM. To appear. Preliminary version in *EUROCRYPT'10*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, 2010.

[16] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-LWE cryptography. In *EUROCRYPT'13*, volume 7881 of *Lecture Notes in Computer Science*, pages 35–54, 2013. Available at `http://eprint.iacr.org/2013/293`.

[17] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, 2011.

[18] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. Designs, Codes and Cryptography. Springer. To appear. DOI 10.1007/s10623-012-9720-4. Available at http://eprint.iacr.org/2011/133.

[19] Lawrence C. Washington. *Introduction to Cyclotomic Fields*, volume 83 of *Graduate Texts in Mathematics*. Springer, 1996.

Bootstrapping BGV Ciphertexts with a Wider Choice of p and q

E. Orsini, J. van de Pol and N.P. Smart

Dept. Computer Science,
University of Bristol,
United Kingdom.
{Emmanuela.Orsini,Joop.VandePol}@bristol.ac.uk,
{nigel}@cs.bris.ac.uk

Abstract. We describe a method to bootstrap a packed BGV ciphertext which does not depend (as much) on any special properties of the plaintext and ciphertext moduli. Prior "efficient" methods such as that of Gentry et al (PKC 2012) required a ciphertext modulus q which was close to a power of the plaintext modulus p. This enables our method to be applied in a larger number of situations. Also unlike previous methods our depth grows only as $O(\log p + \log\log q)$ as opposed to the $\log q$ of previous methods. Our basic bootstrapping technique makes use of a representation of the group $\mathbb{Z}_q^+$ over the finite field $\mathbb{F}_p$ (either based on polynomials or elliptic curves), followed by polynomial interpolation of the reduction mod p map over the coefficients of the algebraic group.
This technique is then extended to the full BGV packed ciphertext space, using a method whose depth depends only logarithmically on the number of packed elements. This method may be of interest as an alternative to the method of Alperin-Sheriff and Peikert (CRYPTO 2013). To aid efficiency we utilize the ring/field switching technique of Gentry et al (SCN 2012, JCS 2013).

1 Introduction

Since the invention of Fully Homomorphic Encryption (FHE) by Gentry in 2009 [14,15], one of the main open questions in the field has been how to "bootstrap" a Somewhat Homomorphic Encryption (SHE) scheme into a FHE scheme. Recall an SHE scheme is one which can evaluate circuits of a limited multiplicative depth, whereas an FHE scheme is one which can evaluate circuits of arbitrary depth. Gentry's bootstrapping technique is the only known way of obtaining unbounded FHE.

The ciphertexts of all known SHE schemes include some noise to ensure security, and unfortunately this noise grows as more and more homomorphic operations are performed, until it is so large that the ciphertext will no longer decrypt correctly. In a nutshell, bootstrapping "refreshes" a ciphertext that can not support any further homomorphic operation by homomorphically decrypting it, and obtaining in this way a new encryption of the some plaintext, but with smaller noise. This is possible if the underlying SHE scheme has enough homomorphic capacity to evaluate its own decryption algorithm. Bootstrapping is computationally very expensive and it represents the main bottleneck in FHE constructions.

Several SHE schemes, with different bootstrapping procedures, have been proposed in the past few years [1,2,4,6,7,8,14,15,10,18,19,32]. The most efficient are ones which allow SIMD style operations, by packing a number of plaintext elements into independent "slots" in the plaintext space. The most studied of such "SIMD friendly" schemes being the BGV scheme [5] based on the Ring-LWE Problem [25].

Prior Work on Bootstrapping. In *almost all* the SHE schemes supporting bootstrapping, decryption is performed by evaluating some linear function D, dependent on the ciphertext c, on the secret key $\mathfrak{sk}$ modulo some integer q, and then reducing the result modulo some prime p, i.e. $\mathsf{dec}(c, \mathfrak{sk}) = ((D_C(\mathfrak{sk}) \mod q) \mod p)$. Given an encryption of the secret key, bootstrapping consists in evaluating the above decryption formula homomorphically. One can divide the bootstrapping of all efficient currently known SHE schemes into three distinct sub-problems.

1. The first problem is to homomorphically evaluate the reduction (mod p)-map on the group $\mathbb{Z}_q^+$ (see Fig. 1), where for the domain one takes representatives centered around zero. To do this the group $\mathbb{Z}_q^+$ is first mapped to a set $\mathbb{G}$ in which one can perform operations native to the homomorphic cryptosystem. In other words we first need to specify a *representation*, $\mathsf{rep} : \mathbb{Z}_q^+ \longrightarrow \mathbb{G}$, which takes an integer in the range $(-q/2, \ldots, q/2]$ and maps it to the set $\mathbb{G}$. The group operation on $\mathbb{Z}_q^+$ needs to induce a group operation on $\mathbb{G}$ which can be evaluated homomorphically by the underlying SHE scheme. Then we describe the induced map $\mathsf{red} : \mathbb{G} \longrightarrow \mathbb{Z}_p$ as a algebraic operation, which can hence be evaluated homomorphically.
2. The second problem is to encode the secret key in a way that one can publicly, using a function dec-eval (decryption evaluation), create a set of ciphertexts which encrypt the required input to the function red.
3. And thirdly one needs a method to extend this to packed ciphertexts.

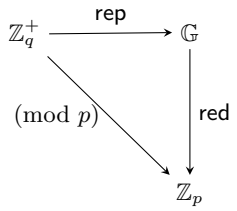

Fig. 1.

To solidify ideas we now expand on these problems in the context of the BGV scheme [5]. Recall for BGV we have a set of $L + 1$ moduli, corresponding to the levels of the scheme, $q_0 < q_1 < \ldots < q_L$, and a (global) ring R, which is often the ring of integers of a cyclotomic number field. We let p denote the (prime) plaintext modulus, i.e. the plaintexts will be elements in R_p (the localisation of R at the prime p), and to ease

notation we set $q = q_0$. The secret key $\mathfrak{sk}$ is a small element in R. A "fresh" ciphertext encrypting $\mu' \in R_p$ is an element $\mathfrak{ct}' = (c'_0, c'_1)$ in $R^2_{q_L}$ such that

$$(c'_0 + \mathfrak{sk} \cdot c'_1 \pmod{q_L}) \pmod{p} = \mu'.$$

After the evaluation of L levels of multiplication one obtains a ciphertext $\mathfrak{ct} = (c_0, c_1)$ in R^2_q, encrypting a plaintext μ, such that

$$(c_0 + \mathfrak{sk} \cdot c_1 \pmod{q}) \pmod{p} = \mu.$$

At this point to perform further calculations one needs to bootstrap, or recrypt, the ciphertext to one of a higher level.

Assume for the moment that each plaintext only encodes a single element of $\mathbb{Z}_p$, i.e. each plaintext is a constant polynomial in polynomial basis for R_p. To perform bootstrapping we need to place a "hint" in the public key $\mathfrak{pk}$ (usually an encryption of $\mathfrak{sk}$ at level L), which allows the following operations. Firstly, we can evaluate homomorphically a function dec-eval which takes $\mathfrak{ct}$ and the "hint", and outputs a representation of the $\mathbb{Z}_q$ element corresponding to the constant term of the element $c_0 + \mathfrak{sk} \cdot c_1 \pmod{q}$. This representation is an encryption of an element in $\mathbb{G}$, i.e. dec-eval also evaluates the rep map as well as the decryption map. Then we apply, homomorphically, the function red to this representation to obtain a fresh encryption of the plaintext. Since to homomorphically evaluate red we need the input to red to be defined over the plaintext space, this means the representation of $\mathbb{Z}_q$ must be *defined over* $\mathbb{F}_p$. One is then left with the task of extending such a procedure to packed ciphertexts.

In the original bootstrapping technique of Gentry [15], implemented in [16], the function dec-eval is obtained from a process of bit-decomposition. Thus the representation $\mathbb{G}$ of $\mathbb{Z}_q$ is the bit-representation of an integer in the range $(-q/2, \ldots, q/2]$, i.e. we use a representation defined over $\mathbb{F}_2$. The function to evaluate red is then the circuit which performs reduction modulo p. The extension of this technique to packed ciphertexts, in the context of the Smart–Vercauteren SIMD optimisations [29] of Gentry's SHE scheme, was given in [30]. Due to the use of bit-decomposition techniques this method is mainly suited to the case of $p = 2$, although one can extend it to other primes by applying a p-adic decomposition and then using an arithmetic circuit to evaluate the reduction modulo p map.

In [18] the authors present a bootstrapping technique, primarily targeted at the BGV scheme, which does away with the need for evaluating the "standard" circuit for the reduction modulo p map. This is done by choosing q close to a power of p, i.e. one selects $q = p^t \pm a$ for some t and a small value of a, typically $a \in \{-1, 1\}$. The paper [18] expands on this idea for the case of $p = 2$, but the authors mention it can be clearly extended to arbitrary p. The advantage is that the mapping red can now be expressed as algebraic formulae; in fact formulae of multiplicative depth $\log_2 q$. The operation dec-eval obtains the required representation for $\mathbb{Z}_q$ by mapping it into $\mathbb{Z}_{p^{t+1}}$. The resulting technique requires the extension of the modulus of the plaintext ring to p^{t+1} (for which all the required properties of R_p carry over, assuming that p does not ramify). The extension to packed ciphertexts is performed using an elaborate homomorphic evaluation of the Fourier Transform.

To enable the faster evaluation of this Fourier Transform step from [18], a method for ring/field switching is presented in [17]. The technique of ring/field switching also enables general improvements in efficiency as ciphertext noise grows. This enables the ring R to be changed to a sub-ring S (both for the ciphertext and plaintext spaces). In [1] this use of field switching is combined with the red map from [18] to obtain an asymptotically efficient bootstrapping method for BGV style SHE schemes; although the resulting technique does not fully map to our blueprint, as $q = p^v$ for some value of v. In [28] this method is implemented, with surprisingly efficient runtimes, for the case of plaintext space $\mathbb{F}_2$; i.e. $p = 2$ and no plaintext SIMD-packing is supported.

In another line of work, the authors of [2] and [8] present a bootstrapping technique for the GSW [21] homomorphic encryption scheme. The GSW scheme is one based on matrices, and this property is exploited in [2] by taking a matrix representation of $\mathbb{Z}_q$ and then expressing the map red via a very simple algebraic relationship on the associated matrices. In particular the authors represent elements of $\mathbb{Z}_q$ by matrices (of some large dimension) over $\mathbb{F}_p$.

Thus we see *almost all* bootstrapping techniques require us to come up with a representation $\mathbb{G}$ of $\mathbb{Z}_q$ for which there is an algebraic method over $\mathbb{F}_p$ to evaluate the induced mapping red, from the said representation of $\mathbb{Z}_q$, to $\mathbb{Z}_p$. Since SHE schemes usually homomorphically have add and multiply operations as their basic homomorphic operations, this implies we are looking for representations of $\mathbb{Z}_q^+$ as a subgroup of an algebraic group over $\mathbb{F}_p$.

Our Contribution. We return to consider the Ring-LWE based BGV scheme, and we present a new bootstrapping technique with small depth growth, compared with previous methods, and which supports a larger choice of p and q. Instead of concentrating on the case of plaintext moduli p such that a power of p is close to q, we look at a much larger class of plaintext moduli. Recall the most efficient prior technique, based on [1] and [18], requires a method whose multiplicative depth is $O(\log q)$, and for which q is close to a power of p. As p increases the ability to select a suitable modulus q which is both close to a power of p, is of the correct size for most efficient implementation (i.e. the smallest needed to ensure security), and has other properties related to efficiency (i.e. the ring R_q has a double-CRT representation as in [20]) diminishes.

To allow a wider selection for p we utilize two "new" (for bootstrapping) representations of the ring $\mathbb{Z}_q$, in much the same way as [2] used an $\mathbb{F}_p$-matrix representation (a.k.a. a linear algebraic group) of $\mathbb{Z}_q^+$. The first one, used for much of this paper for ease of presentation, is based on a polynomial representation for $\mathbb{Z}_q^+$ over $\mathbb{F}_p$, the second one (which is less efficient but allows a greater freedom in selecting q) is based on a representation via elliptic curves. The evaluation of the mapping red using these representations can then be done in expected multiplicative depth $O(\log p + \log\log q)$, i.e. a much shallower circuit than used in prior works, using polynomial interpolation of the red map over the coefficients of the algebraic group.

To ensure this method works, and is efficient, we do not have completely free reign in selecting q for the first polynomial representation. Whilst [18] required $q = p^t \pm a$, for a small value of a, we instead will require that q divides

$$\mathsf{lcm}\left(p^{k_1} - 1, \ldots, p^{k_t} - 1\right),$$

for some pairwise co-prime values k_i. Even with this restriction, the freedom on selecting q is much greater than for the method in [18], especially for large values of p. In the second representation, described in Section 7, we simply need to find elliptic curves over $\mathbb{F}_{p^{k_i}}$ whose group order is divisible by e_i where $\prod e_i = q$. For the elliptic curve based version we do not need pairwise co-prime values of k_i. Indeed on setting $t = 1$ we simply need one curve $E(\mathbb{F}_{p^{k_1}})$ whose group order is divisible by q, which is highly likely to exist, since $p \ll q$, by the near uniform distribution of elliptic curve group orders in the Hasse interval.

Note also that, in the polynomial representation, one does not have complete freedom on selecting the k_i values. If we let $E = \sum k_i$ and $M = \frac{1}{2}\sum k_i \cdot (k_i + 1)$ then the depth of the circuit (which is approximately $\log_2 \log_2 q - \log_2 \log_2 E$) to evaluate red will decrease as E grows, but the number of multiplications required, which is a monotonically increasing function of M, will increase. Note, we can asymptotically make $M = O(\sum k_i \cdot \log k_i)$ using FFT techniques, or $M = O(\sum k_i^{1.58})$ using Karatsuba based techniques, but in practice the k_i will be too small to make such optimization fruitful. For the elliptic curve based version we replace the above E by $E + 1$ and we replace M by a constant multiple of M. However, the depth required by our elliptic curve based version increases.

Our method permits to bootstrap a certain number of packed ciphertexts in parallel, using a form of p-adic decomposition and a matrix representation of the ciphertext ring, combined with ring switching. The resulting depth depends only logarithmically on the number of packed ciphertexts.

Overview and paper organization. Here we give a brief overview of the paper. In Section 2 and 3 we recall the basic algebraic background required for our construction, and the BGV SHE scheme from [5], respectively. Typically, the main technical difficult in bootstrapping is to homomorphically evaluate in a efficient way the (mod p)-map on the group $\mathbb{Z}_q^+$. In Section 4 we describe a simple way to evaluate the (mod p)-map using a polynomial representation of the group $\mathbb{G}$ in Fig. 1. In Section 5 we prepare to bootstrap packed ciphertexts and we show how to homomorphically evaluate a product of powers of SIMD vectors. In particular we calculate the depth and the number of multiplications required to compute this operation. Finally, in Section 6 we show how to bootstrap BGV ciphertexts. We use a matrix representation of the product of two elements in a ring and a single ring switching step in such a way that we can bootstrap a number, say C, of packed ciphertexts in one step. We describe the homomorphic evaluation of the decryption equation using the SIMD evaluation of the maps red and rep. Using the calculation of Section 5, we can compute the depth and the number of multiplications necessary to bootstrap C packed ciphertexts in parallel. In Section 7 we give a different instantiation of our method using elliptic curves.

2 Preliminaries

Throughout this work vectors are written using bold lower-case letters, whereas bold upper-case letters are used for matrices. We denote by $M_{a\times b}(K)$ the set of $a \times b$ dimensional matrices with entries in K. For an integer modulus q, we let $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$

denote the quotient ring of integers modulo q, and $\mathbb{Z}_q^+$ its additive group. This notation naturally extends to the localisation R_q of a ring R at q.

2.1 Algebraic Background

Let m be a positive integer we define the mth cyclotomic field to be the field $\mathbb{K} = \mathbb{Q}[X]/\Phi_m(X)$, where $\Phi_m(X)$ is the mth cyclotomic polynomial. $\Phi_m(X)$ is a monic irreducible polynomial over the rational, and $\mathbb{K}$ is a field extension of degree $N = \phi(m)$ over $\mathbb{Q}$ since $\Phi_m(X)$ has degree N. Let ζ_m be an abstract primitive mth roots of unity, we have that $\mathbb{K} \cong \mathbb{Q}(\zeta_m)$ by identifying ζ_m with X. In the same way, let us denote by R the mth cyclotomic ring $\mathbb{Z}[\zeta_m] \cong \mathbb{Z}[X]/\Phi_m(X)$, with "power basis" $\{1, \zeta_m, \ldots, \zeta_m^{N-1}\}$. The complex embeddings of $\mathbb{K}$ are $\sigma_i : \mathbb{K} \to \mathbb{C}$, defined by $\sigma_i(X) = \zeta_m^i$, $i \in \mathbb{Z}_m^*$. In particular $\mathbb{K}$ is Galois over $\mathbb{Q}$ and $\mathrm{Gal}(\mathbb{Q}(\zeta_m)/\mathbb{Q}) \cong \mathbb{Z}_m^*$. As a consequence we can define the $\mathbb{Q}$-linear (field) trace $\mathrm{Tr}_{\mathbb{K}/\mathbb{Q}} : \mathbb{K} \to \mathbb{Q}$ as the sum of the embeddings σ_i, i.e. $\mathrm{Tr}_{\mathbb{K}/\mathbb{Q}}(a) = \sum_{i \in \mathbb{Z}_m^*} \sigma_i(a) \in \mathbb{Q}$. Concretely, these embeddings map ζ_m into each of its conjugates, and they are the only field homomorphisms from $\mathbb{K}$ to $\mathbb{C}$ that fix every element of $\mathbb{Q}$. The *canonical embedding* $\sigma : \mathbb{K} \to \mathbb{C}^N$ is the concatenation of all the complex embeddings, i.e. $\sigma(a) = (\sigma_i(a))_{i \in \mathbb{Z}_m^*}$, $a \in \mathbb{K}$.

Looking ahead, we will use the ring R and its localisation R_q, for some modulus q. Given a polynomial $a \in R$, we denote by $\|\mathbf{a}\|_\infty = \max_{0 \le j \le N-1} |a_j|$ the standard l_∞-norm. All estimates of noise are taken with respect to the *canonical embedding norm* $\|a\|_\infty^{\mathsf{can}} = \|\sigma(a)\|_\infty$, $a \in R$. When considering short elements in R_q, we define short in terms of the following quantity:

$$|a|_q^{\mathsf{can}} = \min\{\|a'\|_\infty^{\mathsf{can}} : a' \in R \text{ and } a' \equiv a \mod q\}.$$

To map from norms in the canonical embedding to norms on the coefficients of the polynomial defining the elements of R, we have $\|\mathbf{a}\|_\infty \le c_m \cdot \|a\|_\infty^{\mathsf{can}}$, where c_m is the *ring constant*. For more details about c_m see [13]. Note, if the dual basis techniques of [26] are used, then one can remove the dependence on c_m. However, for ease of exposition we shall use only polynomial basis in this work.

Let m' be a positive integer such that $m'|m$. As before we define $\mathbb{K}' \cong \mathbb{Q}(\zeta_{m'})$ and $S \cong \mathbb{Z}[\zeta_{m'}]$, such that $\mathbb{K}'$ has degree $n = \phi(m')$ over $\mathbb{Q}$ and $\mathrm{Gal}(\mathbb{K}'/\mathbb{Q}) \cong \mathbb{Z}_{m'}^*$. It is trivial to show that $\mathbb{K}$ and R are a field and a ring extension of $\mathbb{K}'$ and R', respectively, both of dimension N/n. In particular we can see S as a subring of R via the ring embedding that maps $\zeta_{m'} \mapsto \zeta_m^{m/m'}$.

It is a standard fact that if $\mathbb{Q} \subseteq \mathbb{K}' \subseteq \mathbb{K}$ is a tower of number field, then $\mathrm{Tr}_{\mathbb{K}/\mathbb{Q}}(a) = \mathrm{Tr}_{\mathbb{K}'/\mathbb{Q}}(\mathrm{Tr}_{\mathbb{K}/\mathbb{K}'}(a))$, and that all the $\mathbb{K}'$-linear maps $L : \mathbb{K} \to \mathbb{K}'$ are exactly the maps of the form $\mathrm{Tr}_{\mathbb{K}/\mathbb{K}'}(r \cdot a)$, for some $r \in \mathbb{K}$.

2.2 Plaintext Slots

Let p be a prime integer, coprime to m, and R_p the localisation of R at p. The polynomial $\Phi_m(X)$ factors modulo p into $\ell^{(R)}$ irreducible factors, i.e. $\Phi_m(X) \equiv \prod_{i=1}^{\ell^{(R)}} F_i(X) \pmod p$. Each $F_i(X)$ has degree $d^{(R)} = \phi(m)/\ell^{(R)}$, where $d^{(R)}$ is the multiplicative

order of p in $\mathbb{Z}_m^*$. Looking ahead, each of these $\ell^{(R)}$ factors corresponds to a "plaintext slot", i.e.

$$R_p \cong \mathbb{Z}_p[X]/F_1(X) \times \cdots \times \mathbb{Z}_p[X]/F_{\ell^{(R)}}(X) \cong \left(\mathbb{F}_{p^{d^{(R)}}}\right)^{\ell^{(R)}}.$$

More precisely, we have $\ell^{(R)}$ isomorphisms $\psi_i : \mathbb{Z}_p[X]/F_i(X) \to \mathbb{F}_{p^{d^{(R)}}}, i = 1, \ldots, \ell^{(R)}$, that allow to represent $\ell^{(R)}$ plaintext elements of $\mathbb{F}_{p^{(d)}}$ as a single element in R_p. By the Chinese Remainder Theorem, addition and multiplication correspond to SIMD operations on the slots and this allows to process $\ell^{(R)}$ input values at once.

2.3 Ring Switching

As mentioned in the introduction, our technique uses a method for ring/field switching from [17] so as to aid efficiency. We use two different cyclotomic rings R and S such that $S \subseteq R$. This procedure permits to transform a ciphertext $\mathfrak{ct} \in (R_q)^2$ corresponding to a plaintext $\mu \in R_p$ with respect to a secret key $\mathfrak{sk} \in R$, into a ciphertext $\mathfrak{ct}' \in (S_q)^2$ corresponding to a plaintext $\mu' \in S_p$ with respect to a secret key $\mathfrak{sk}' \in S$. The security of this method relies on the hardness of the ring-LWE problem in S ([25]). At a high level the ring switching consists of three steps. Given an input ciphertext $\mathfrak{ct} \in (R_q)^2$:

- First, it switches the secret key; it uses the "classical" key-switching ([6],[5]), getting a ciphertext $\bar{\mathfrak{ct}} \in (R_q)^2$, still encrypting $\mu \in R_p$, but with respect to a secret key $\mathfrak{sk}' \in S$.
- Second, it multiplies $\bar{\mathfrak{ct}}$ by a fixed element $r \in R$, which is determined by a S-linear function $L : R_p \to S_p$ corresponding to the induced projection function $P : \left(\mathbb{F}_{p^{d^{(R)}}}\right)^{\ell^{(R)}} \to \left(\mathbb{F}_{p^{d^{(S)}}}\right)^{\ell^{(S)}}$ (see [17] for details).
- Finally, it applies to $\bar{\mathfrak{ct}}$ the trace function $\mathrm{Tr}_{R/S} : R \to S$. In such a way the output of the ring-switching is a ciphertext $\mathfrak{ct} \in S$ with respect to the secret key $\mathfrak{sk}'$ and encrypting the plaintext $\mu' = L(\mu)$.

We conclude this section noting that, while big-ring ciphertexts correspond to $\ell^{(R)}$ plaintext slots, small-ring ciphertexts only correspond to $\ell^{(S)} \leq \ell^{(R)}$ plaintext slots. The input ciphertexts to our bootstrapping procedure are defined over $(S_q)^2$, and so are of degree n and contain $\ell^{(S)}$ slots. We take $\ell^{(R)}/n$ of these ciphertexts and use the dec-eval map to encode the coefficients of the plaintext polynomials in the slots of a single big-ring ciphertext. Eventually, via ring switching and polynomial interpolation, we return to $\ell^{(R)}/n$ ciphertexts which have been bootstrapped and are at level one (or more). These fresh ciphertexts may be defined over the big ring or the small ring (depending when ring switching occurs). However, our parameter estimates imply that ring switching is best performed at the lowest level possible, and so our bootstrapped ciphertexts will be in the big ring. We could encode all of the slots of the bootstrapped ciphertexts in a big-ring single ciphertext, or not, depending on the application, since slot manipulation is a linear operation.

3 The BGV Somewhat Homomorphic Encryption Scheme

In this section we outline what we need about the BGV SHE scheme [5]. As anticipated in Section 2, we present the scheme with the option of utilizing two rings, and hence at some point we will make use of the ring/field switching procedure from [17]. We first define two rings $R = \mathbb{Z}[X]/F(X)$ and $S = \mathbb{Z}[X]/f(X)$, where $F(X)$ (resp. $f(x)$) is an irreducible polynomial over Z of degree N (resp. n). In practice both $F(X)$ and $f(X)$ will likely be cyclotomic polynomials. We assume that n divides N, and so here is an embedding $\iota : S \longrightarrow R$ which maps elements in S to their appropriate equivalent in R. The map ι can be expressed as a linear mapping on the coefficients of the polynomial representation of the elements in S, to the coefficients of the polynomial representation of the elements in R. In this way we can consider S to be a subring of R.

Let R_q (resp. S_q) denote the localisation of R (resp S) at q, i.e. $\mathbb{Z}_q[X]/F(X)$ (resp. $\mathbb{Z}_q[X]/f(X)$), which can be constructed for any positive integer q. Let p be a prime number, which does not ramify in either R or S. Since the rings are Galois, the ring R_p (resp. S_p) splits into $\ell^{(R)}$ (resp. $\ell^{(S)}$) "slots"; with each slot being a finite field extension of $\mathbb{F}_p$ of degree $d^{(R)} = N/\ell^{(R)}$ (resp. $d^{(S)} = n/\ell^{(S)}$). We make the assumption that n divides $\ell^{(R)}$. This is not strictly necessary but it ensures that we can perform bootstrapping of a single ciphertext with the smallest amount of memory. In fact our method will support the bootstrapping of $\ell^{(R)}/n$ ciphertexts in parallel.

There will be two secret keys for our scheme; depending on whether the ciphertexts/plaintexts are associated with the ring R or the ring S. We denote these secret keys by $\mathfrak{sk}^{(R)}$ and $\mathfrak{sk}^{(S)}$, which are "small" elements in the ring R (resp. S). The modulus $q = q_0 = p_0$ will denote the smallest modulus in the set of BGV levels. Fresh ciphertexts are defined for the modulus $Q = q_L = \prod_{i=0}^{L} p_i$ and live in the ring R_Q^2 (thus at some point we not only perform modulus switching but also ring switching). We assume L_1 levels are associated with the big ring R and L_2 levels are associated with the small ring S, hence $L_1 + L_2 = L$ (level zero is clearly associated with the small ring S, but we do not count it in the number of levels in L_2). Thus we encrypt at level L; perform standard homomorphic operations down to level zero, with a single field switch at level $L_2 + 1$. For ease of analysis we assume no multiplications are performed at level $L_2 + 1$. This means that we can evaluate a depth $L - 1$ circuit.

A ciphertext at level $i > L_2$, encrypting a message $\mu \in R_p$, is a pair $\mathfrak{ct} = (c_0, c_1) \in R_{q_i}^2$, where $q_i = \prod_{j=0}^{i} p_j$, such that

$$\Big(c_0 + \mathfrak{sk}^{(R)} \cdot c_1 \pmod{q_i}\Big) \pmod{p} = \mu.$$

We let $\mathsf{Enc}_{\mathfrak{pk}}(\mu)$ denote the encryption of a message $\mu \in R_p$, this produces a ciphertext at level L. A similar definition holds for ciphertexts at level $i < L_2$, for messages in S_p and secret keys/ciphertexts elements in S_{q_i}. When performing a ring switching operation between levels $L_2 + 1$ and L_2, the $\ell^{(R)}$ plaintext slots, associated with the input ciphertext at level $L_2 + 1$, become associated with $\ell^{(R)}/\ell^{(S)}$ distinct ciphertexts at level L_2.

We want to "bootstrap" a set of BGV ciphertexts. Each of these ciphertexts is a pair $\mathfrak{ct}_j = (c_0^{(j)}, c_1^{(j)}) \in S_q^2$, for $j = 1, \ldots, \ell^{(R)}/n$, such that

$$\left(c_0^{(j)} + \mathfrak{sk}^{(S)} \cdot c_1^{(j)} \pmod q\right) \pmod p = \mu_j, \text{ for } j = 1, \ldots, \ell^{(R)}/n.$$

4 Evaluating the Map red ∘ rep : $\mathbb{Z}_q^+ \longrightarrow \mathbb{F}_p$ (Simple Version)

As explained in the introduction at the heart of most bootstrapping procedures is a method to evaluate the induced mapping $\mathsf{red} \circ \mathsf{rep} : \mathbb{Z}_q^+ \longrightarrow \mathbb{F}_p$. In this section we present our simpler technique for doing this based on polynomials over $\mathbb{F}_p$, in Section 7 we present a more general (and complicated in terms of depth) technique based on elliptic curves. The key, in this and in all techniques, is to find a representation $\mathbb{G}$ for $\mathbb{Z}_q^+$ for which the reduction modulo p map can be evaluated algebraically over $\mathbb{F}_p$. This means that the representation of $\mathbb{Z}_q$ must defined over $\mathbb{F}_p$. Prior work has looked at the bit-representation (when $p = 2$), the p-adic representation and a matrix representation; we use a polynomial representation.

We select a coprime factorization $q = \prod_{i=1}^t e_i$ (with the e_i not necessarily prime, but pairwise coprime), such that e_i divides $p^{k_i} - 1$ for some k_i. Since $\mathbb{F}^*_{p^{k_i}}$ is cyclic we know that $\mathbb{F}^*_{p^{k_i}}$ has a subgroup of order e_i. We fix a polynomial representation of $\mathbb{F}_{p^{k_i}}$, i.e. an irreducible polynomial $f_i(x)$ of degree k_i such that $\mathbb{F}_{p^{k_i}} = \mathbb{F}_p[x]/f_i(x)$. Let $g_i \in \mathbb{F}_{p^{k_i}}$ denote a fixed element of order e_i in $\mathbb{F}_{p^{k_i}}$.

By the Chinese Remainder Theorem we therefore have a group embedding

$$\mathsf{rep} : \begin{cases} \mathbb{Z}_q^+ \longrightarrow \mathbb{G} = \prod_{i=1}^t \mathbb{F}^*_{p^{k_i}} \\ a \longmapsto (g_1^{a_1}, \ldots, g_t^{a_t}) \end{cases} \tag{1}$$

where $a_i = a \pmod{e_i}$. Without loss of generality we can assume that the k_i are also coprime, by modifying the decomposition of q into coprime e_is. Given this group representation of $\mathbb{Z}_q^+$ in $\mathbb{G}$, addition in $\mathbb{Z}_q^+$ translates into multiplication in $\mathbb{G}$. With one addition in $\mathbb{Z}_q^+$ translating into $M = \frac{1}{2}\sum_{i=1}^t k_i \cdot (k_i + 1)$ multiplications in $\mathbb{F}_p$ (and a comparable number of additions; assuming school book multiplication is used). Each element in the image of rep requires $E = \sum_{i=1}^t k_i$ elements in $\mathbb{F}_p$ to represent it.

There will be a map $\mathsf{red} : \mathbb{G} \to \mathbb{F}_p$, such that $\mathsf{red} \circ \mathsf{rep}$ is the reduction modulo p map; and red can be defined by *algebraically* from the coefficient representation of $\mathbb{G}$ to $\mathbb{F}_p$. Here algebraically refers to algebraic operations over $\mathbb{F}_p$. An arbitrary algebraic expression on E variables of degree d will contain ${}^{d+E}C_d$ terms. Thus, by interpolating, we expect the degree d of the map red to be the smallest d such that ${}^{d+E}C_d > q$, which means we expect we expect $d \approx E \cdot (2^{\log(q)/E} - 1)$. Thus the larger E is, the smaller d will be. This interpolating function needs to be created once and for all for any given set of parameters, thus we ignore the cost in generating it in our analysis.

The algebraic circuit which implements the map red can hence be described as a circuit of depth $\lceil \log_2 d \rceil$ which requires $D(E, d) = {}^{E+d}C_d - (E + 1)$ multiplications (corresponding to the number of distinct monomials in E variables of degree between two and d). In particular, by approximating $E \approx \log_2(q)/\log_2(p)$, we obtain that the

circuit implementing the map red has depth $\lceil \log_2 d \rceil = \log_2(p-1) + \log_2(\log_2(q)) - \log_2(\log_2(p))$.

We pause to note the following. By selecting a large finite field it would appear at first glance that one can reduce our degree d even further. This however comes at the cost of having more terms, i.e. a larger value of E. This in turn increases the overall complexity of the method (i.e. the number of multiplications needed) but not the depth.

5 A Product of Powers of SIMD Vectors

Before proceeding with our method to turn the above methodology for reduction modulo p into a bootstrapping method for our set of BGV ciphertexts, we first examine how to homomorphically compute the following function

$$\mathbf{v} \cdot \prod_{k=0}^{\lambda} \mathbf{v}_k^{\mathbf{M}_k},$$

where each $\mathbf{v}$ and $\mathbf{v}_k$, $k = 0, \ldots, \lambda$, represents a set of E ciphertexts, each of which encode (in a SIMD manner) $\ell^{(R)}$ elements in $\mathbb{F}_p$. The multiplication of two such sets of E ciphertexts is done with respect to the multiplication operation in $\mathbb{G}$, and thus requires M homomorphic multiplications (this is for our simple variation of red, for the variant based on elliptic curve the number of ciphertexts and the complexity of the group operation in $\mathbb{G}$ increase a little). The values $\mathbf{M}_k$ are matrices in $M_{\ell^{(R)} \times \ell^{(R)}}(\mathbb{F}_p)$. By the notation $\mathbf{u} = \mathbf{v}^{\mathbf{M}}$, where $\mathbf{M} = (m_{i,j})$, we mean the vector with components

$$u_i = \prod_{j=1}^{\ell^{(R)}} v_j^{m_{i,j}}, \quad i \in \{1, \ldots, \ell^{(R)}\}.$$

Notice that each u_i and v_j is a vector of E elements in $\mathbb{F}_p$ representing a single element in $\mathbb{G}$. In what follows we divide this operation into three sub-procedures and compute the number of multiplications, and the depth required, to evaluate the function.

5.1 SIMD Raising of an Encrypted Vector to the Power of a Public Vector

The first step is to take a vector $\mathbf{v}$ which is the SIMD encryption of E sets of $\ell^{(R)}$ elements in $\mathbb{F}_p$, i.e. it represents $\ell^{(R)}$ elements in $\mathbb{G}$. We then raise $\mathbf{v}$ to the power of some public vector $\mathbf{c} = (c_1, \ldots, c_{\ell^{(R)}})$, i.e. we want to compute

$$\mathbf{x} = \mathbf{v}^{\mathbf{c}}.$$

In particular $\mathbf{v}$ actually consists of E vectors each with $\ell^{(R)}$ components in their slots. We write

$$\mathbf{v} = (\mathbf{v}_{1,0}, \ldots, \mathbf{v}_{1,k_1-1}, \ldots, \mathbf{v}_{t,0}, \ldots, \mathbf{v}_{t,k_t-1}).$$

Note, multiplying such a vector by another vector of the same form requires M homomorphic multiplications and depth 1. We first write

$$\mathbf{c} = \mathbf{c}_0 + 2 \cdot \mathbf{c}_1 + \ldots + 2^{\lceil \log_2 p \rceil} \cdot \mathbf{c}_{\lceil \log_2 p \rceil},$$

where $\mathbf{c}_i \in \{0,1\}^{\ell^{(R)}}$. We let $\mathbf{c}_i^*$ denote the bitwise complement of $\mathbf{c}_i$. Thus to compute $\mathbf{x} = \mathbf{v}^{\mathbf{c}}$ we use the following three steps:

Step 1: Compute $\mathbf{v}^{2^i}$ for $i = 1, \ldots, \lceil \log_2 p \rceil$, by which we mean every element in $\mathbf{v}$ is raised to the power 2^i. This requires $\lceil \log_2 p \rceil \cdot M$ homomorphic multiplications and depth $\lceil \log_2 p \rceil$.

Step 2: For $i \in \{0, \ldots, \lceil \log_2 p \rceil\}$, $j \in \{1, \ldots, t\}$ and $k = \{0, \ldots, k_t - 1\}$ compute,

$$\mathbf{w}_{j,k}^{(i)} = \begin{cases} \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{c}_i) \cdot \mathbf{v}_{j,k}^{2^i} & k \neq 0, \\ \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{c}_i) \cdot \mathbf{v}_{j,k}^{2^i} + \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{c}_i^*) & k = 0. \end{cases}$$

Where $\mathsf{Enc}_{\mathfrak{pk}}(\mathbf{c}_i)$ means encrypt the vector $\mathbf{c}_i$ so that the jth component of $\mathbf{c}_i$ is mapped to the jth plaintext slot of the ciphertext. The above procedure selects the values which we want to include in the final product. This involves a homomorphic multiplication by a constant in $\{0, 1\}$ and the homomorphic addition of a constant in $\{0, 1\}$ for each entry, and so is essentially fast (and moderately bad on the noise, so we will ignore this and call it depth $1/2$).

Step 3: We now compute $\mathbf{x}$ as

$$\mathbf{x} = \prod_{i=0}^{\lceil \log_2 p \rceil} \mathbf{w}^{(i)},$$

where we think of $\mathbf{w}^{(i)}$ as a vector of E SIMD encryptions. This step (assuming a balanced multiplication tree) requires depth $\lceil \log_2 \lceil \log_2 p \rceil \rceil$ and $M \cdot \lceil \log_2 p \rceil$ multiplications.

Executing all three steps above therefore requires a depth of $\frac{1}{2} + \lceil \log_2 p \rceil + \lceil \log_2 \lceil \log_2 p \rceil \rceil$, and $2 \cdot M \cdot \lceil \log_2 p \rceil$ multiplications.

5.2 Computing $\mathbf{u} = \mathbf{v}^M$

Given the previous subsection, we can now evaluate $u_i = \prod_{j=1}^{\ell^{(R)}} v_j^{m_{i,j}}, i = 1, \ldots, \ell^{(R)}$, where $\mathbf{v}$ is a SIMD vector consisting of E vectors encoding $\ell^{(R)}$ elements, as is the output $\mathbf{u}$. For this we use a trick for systolic matrix-vector multiplication in [22], but converted into multiplicative notation.

We write the matrix $\mathbf{M}$ as $\ell^{(R)}$ SIMD vectors $\mathbf{d}_i$, for $i = 1, \ldots, \ell^{(R)}$, so that $\mathbf{d}_{i,j} = m_{j,(j+i-1) \pmod{\ell^{(R)}}}$ for $j = 1, \ldots, \ell^{(R)}$. We let $\mathbf{v} \lll i$ denote the SIMD vector $\mathbf{v}$ rotated left i positions (with wrap around). Since $\mathbf{v}$ actually consists of E SIMD vectors this can be performed using time proportional to E multiplications, but with no addition to the overall depth (it is an expensive in terms of time, but cheap in terms of noise. See the operations in Table 1 of [22]).

Step 1: First compute, for $i = 1, \ldots, \ell^{(R)}$,

$$\mathbf{x}_i = (\mathbf{v} \lll (i-1))^{\mathbf{d}_i}$$

using the method previously described in Subsection 5.1. This requires a depth of $\frac{1}{2} + \lceil \log_2 p \rceil + \lceil \log_2 \lceil \log_2 p \rceil \rceil$, and essentially $\ell^{(R)} \cdot (E + 2 \cdot M \cdot \lceil \log_2 p \rceil)$ multiplications.

Step 2: All we need now do is compute

$$\mathbf{u} = \prod_{i=1}^{\ell^{(R)}} \mathbf{x}_i.$$

This requires (assuming a balanced multiplication tree) a depth of $\lceil \log_2 \ell^{(R)} \rceil$ and $\ell^{(R)}$ multiplications in $\mathbb{G}$.

Thus far, for the operations in Subsection 5.1 and this subsection we have used a total depth of $\frac{1}{2} + \lceil \log_2 \ell^{(R)} \rceil + \lceil \log_2 p \rceil + \lceil \log_2 \lceil \log_2 p \rceil \rceil$ and a cost of $\ell^{(R)} \cdot (M + E + 2 \cdot M \cdot \lceil \log_2 p \rceil)$ multiplications.

5.3 Computing $\mathbf{v} \cdot \prod_{k=0}^{\lambda} \mathbf{v}_k^{M_k}$

To evaluate our required output we need to execute the above steps λ times, in order to obtain the elements which we then multiply together. Thus in total we have a depth of

$$\frac{1}{2} + \lceil \log_2 \ell^{(R)} \rceil + \lceil \log_2 p \rceil + \lceil \log_2 \lceil \log_2 p \rceil \rceil + \lceil \log_2 \lambda \rceil$$

and a cost of

$$\lambda \cdot \Big(M + \ell^{(R)} \cdot (M + E + 2 \cdot M \cdot \lceil \log_2 p \rceil) \Big)$$

multiplications.

6 Bootstrapping a Set of Ciphertexts

To perform our bootstrapping operation we introduce another representation, this time more standard. This is the matrix representation of the ring S_q. Since S_q can be considered a vector space over $\mathbb{Z}_q$ by the usual polynomial embedding, we can associate an element a to its coefficient vector $\mathbf{a}$. We can also associate an element b to a $n \times n$ matrix $\mathbf{M}_b$ over $\mathbb{Z}_q$ such that the vector

$$\mathbf{c} = \mathbf{M}_b \cdot \mathbf{a}$$

is the coefficient vector of c where $c = a \cdot b$. This representation, which associates an element in S_q to a matrix, is called the matrix representation.

Recall we want to bootstrap $\ell^{(R)}/n$ ciphertexts in one go. We also recall the maps red and rep from Section 4 and define $\tau = \mathsf{red} \circ \mathsf{rep}$ to be the reduction modulo p map

on $\mathbb{Z}_q^+$. To do this we can first extend rep and τ to the whole of S_q^+ by linearity, with images in $\mathbb{G}^n$ and $\mathbb{F}_p^n$ respectively. Similarly, we can extend rep and τ to $S_q^{\ell^{(R)}/n}$ to obtain maps $\overline{\mathsf{rep}} : (S_q^+)^{\ell^{(R)}/n} \longrightarrow \mathbb{G}^{\ell^{(R)}}$ and $\overline{\tau} : (S_q^+)^{\ell^{(R)}/n} \longrightarrow \mathbb{F}_p^{\ell^{(R)}}$, as in Section 4. Again this induces a map $\overline{\mathsf{red}}$, which is just the SIMD evaluation of red on the image of $\overline{\mathsf{rep}}$ in $\mathbb{G}^{\ell^{(R)}}$. We let $\overline{\mathsf{rep}}_{j,i}$ denote the restriction of $\overline{\mathsf{rep}}$ to the $(i-1)$th coefficient of the j-th S_q component, for $1 \le i \le n$ and $1 \le j \le \ell^{(R)}/n$.

We can then rewrite the decryption equation of our $\ell^{(R)}/n$ ciphertexts as

$$\begin{aligned}\Big(\Big(c_0^{(j)} + \mathfrak{sk}^{(S)} \cdot c_1^{(j)} \pmod q\Big) \pmod p\Big)_{j=1}^{\ell^{(R)}/n} \\ = \overline{\mathsf{red}}\left(\overline{\mathsf{rep}}\left(c_0^{(1)} + \mathfrak{sk}^{(S)} \cdot c_1^{(1)}, \ldots \right.\right. \\ \left.\left. \ldots, c_0^{(\ell^{(R)}/n)} + \mathfrak{sk}^{(S)} \cdot c_1^{(\ell^{(R)}/n)}\right)\right) \\ = \overline{\mathsf{red}}\left(\overline{\mathsf{rep}}\left(\mathbf{x}\right)\right),\end{aligned}$$

where $\mathbf{x}$ is the vector consisting of S_q elements $c_0^{(j)} + \mathfrak{sk}^{(S)} \cdot c_1^{(j)}$, for $j = 1, \ldots, \ell^{(R)}/n$. Thus, if we can compute $\overline{\mathsf{rep}}(\mathbf{x})$, then to perform the bootstrap we need only evaluate (in $\ell^{(R)}$-fold SIMD fashion) the arithmetic circuit of multiplicative depth $\lceil \log_2 d \rceil$ representing $\overline{\mathsf{red}}$. Since we have enough slots, $\ell^{(R)}$, in the large plain text ring, we are able to do this homomorphically on fully packed ciphertexts. The total number of monomials in the arithmetic circuit (i.e. the multiplications we would need to evaluate $\overline{\mathsf{red}}$) being $D(E, d)$.

6.1 Homomorphically Evaluating $\overline{\mathsf{rep}}(\mathbf{x})$

We wish to homomorphically evaluate $\overline{\mathsf{rep}}(\mathbf{x})$ such that the output is a set of E ciphertexts and if we took the $i + (j-1) \cdot \ell^{(R)}/n$th slot of each plaintext we would obtain the E values which represent $\overline{\mathsf{rep}}_{j,i}(\mathbf{x})$. Let $\lambda = \lceil \log q / \log p \rceil$. We add to the public key of the SHE scheme the encryption of $\overline{\mathsf{rep}}(p^k \cdot \mathfrak{sk}^{(S)}, \ldots, p^k \cdot \mathfrak{sk}^{(S)})$ for $k = 0, \ldots, \lambda$ (where each component is copied $\ell^{(R)}/n$ times). For a given k this is a set of E ciphertexts, such that if we took the $i + (j-1) \cdot \ell^{(R)}/n$th slot of each plaintext we would obtain the E values which represent $\overline{\mathsf{rep}}_{j,i}(p^k \cdot \mathfrak{sk}^{(S)})$. Let the resulting vector of ciphertexts be denoted $\mathfrak{ct}_k$, for $k = 1, \ldots, \lambda$, where $\mathfrak{ct}_k$ is a vector of length E.

Let $\mathbf{M}_{c_1^{(j)}}$ be the matrix representation of the second ciphertext component $c_1^{(j)}$ of the j-th ciphertext that we want to bootstrap. We write

$$\mathbf{M}_{c_1^{(j)}} = \sum_{k=0}^{\lambda} p^k \cdot \mathbf{M}_1^{(j,k)}$$

where $\mathbf{M}_1^{(j,k)}$ is a matrix with coefficients in $\{0, \ldots, p-1\}$. We then have that

$$\begin{aligned} c_0^{(j)} + \mathfrak{sk}^{(S)} \cdot c_1^{(j)} &= c_0^{(j)} + \sum_{k=0}^{\lambda} \left(p^k \cdot \mathbf{M}_1^{(j,k)} \cdot \underline{\mathfrak{sk}}^{(S)} \right) \\ &= c_0^{(j)} + \sum_{k=0}^{\lambda} \left(\mathbf{M}_1^{(j,k)} \cdot (p^k \cdot \underline{\mathfrak{sk}}^{(S)}) \right), \end{aligned}$$

where $\underline{\mathfrak{sk}}^{(S)}$ is the vector of coefficients of the secret key $\mathfrak{sk}^{(S)}$.

We let $\mathbf{M}_1^{(k)} = \bigoplus_{j=1}^{\ell^{(R)}/n} \mathbf{M}_1^{(j,k)} = \mathbf{diag}(\mathbf{M}_1^{(1,k)}, \ldots, \mathbf{M}_1^{(\ell^{(R)}/n,k)})$. We now apply $\overline{\mathsf{rep}}$ to both sides, which means we need to compute homomorphically the ciphertext which represents

$$\overline{\mathsf{rep}} \left(c_0^{(1)}, \ldots, c_0^{(\ell^{(R)}/n)} \right) \cdot \prod_{k=0}^{\lambda} \overline{\mathsf{rep}} \left(p^k \cdot \underline{\mathfrak{sk}}^{(S)}, \ldots, p^k \cdot \underline{\mathfrak{sk}}^{(S)} \right)^{\mathbf{M}_1^{(k)}}.$$

We are thus in the situation described in Section 5. Thus the homomorphic evaluation of $\overline{\mathsf{rep}}(\mathbf{x})$ requires a depth of

$$\frac{1}{2} + \lceil \log_2 \ell^{(R)} \rceil + \lceil \log_2 p \rceil + \lceil \log_2 \lceil \log_2 p \rceil \rceil + \lceil \log_2 \lambda \rceil$$

and

$$\lambda \cdot \left(M + \ell^{(R)} \cdot (M + E + 2 \cdot M \cdot \lceil \log_2 p \rceil) \right)$$

multiplications.

6.2 Repacking

At this point in the bootstrapping procedure (assuming for simplicity that a ring switch has not occured) we have a single ciphertext $\mathfrak{ct}$ whose $\ell^{(R)}$ slots encode the coefficients (over the small ring) of the $\ell^{(R)}/n$ ciphertexts that we are bootstrapping. Our task is now to extract these coefficients to produce a ciphertext (or set of ciphertexts) which encode the same data. Effectively this is the task of performing $\ell^{(R)}/n$ inverse Fourier transforms (a.k.a interpolations) over S in parallel, and then encoding the result as elements in R via the embedding $\iota : S \longrightarrow R$.

There are a multitude of ways of doing this step (bar performing directly an inverse FFT algorithm), for example the general method of Alperin-Sheriff and Peikert [1] could be applied. This makes the observation that the FFT to a vector of Fourier coefficients $\mathbf{x}$ is essentially applying a linear operation, and hence we can compute it by taking the trace of a value $\alpha \cdot \mathbf{x}$ for some fixed constant α.

We select a more naive, and simplistic approach. Suppose $\mathbf{x}$ is the vector which is encoded by the input ciphertext. We first homomorphically compute

$$\mathbf{b}_1, \ldots, \mathbf{b}_{\ell^{(R)}} = \mathsf{replicate}(\mathbf{x}).$$

Where $\mathsf{replicate}(\mathbf{x})$ is the Full Replication algorithm from [22]. This produces $\ell^{(R)}$ ciphertexts, the ith of which encodes the constant polynomial over R_p equal to the i slot in $\mathbf{x}$. In [22] this is explained for the case where $\ell^{(R)} = N$, but the method clearly works when $\ell^{(R)} < N$. The method requires time $O(\ell^{(R)})$ and depth $O(\log\log \ell^{(R)})$.

Given the output $\mathbf{b}_1, \ldots, \mathbf{b}_{\ell^{(R)}}$, which encode the coefficients of the $\ell^{(R)}/n$ original plaintext vectors, we can now apply ι (which recall is a linear map) to obtain *any* linear function of the underlying plaintexts. For example we could produce $\ell^{(R)}/n$ ciphertexts each of which encodes one of the original plaintexts, or indeed a single ciphertext which encodes all of them.

So putting all of the sub-procedures for bootstrapping together, we find that we can bootstrap $\ell^{(R)}/n$ ciphertexts in parallel using a procedure of depth of

$$\lceil \log_2 d \rceil + \frac{1}{2} + \lceil \log_2 \ell^{(R)} \rceil + \lceil \log_2 p \rceil + \lceil \log_2 \lceil \log_2 p \rceil \rceil + \lceil \log_2 \lambda \rceil + O(\log_2 \log_2 \ell^{(R)})$$

and a cost of

$$D(E, d) + \lambda \cdot \Big(M + \ell^{(R)} \cdot (M + E + 2 \cdot M \cdot \lceil \log_2 p \rceil) \Big) + O(\ell^{(R)})$$

multiplications, where $d \approx (\log_2 q) \cdot (p-1)/(\log_2 p)$, $E = \sum_{i=1}^{t} k_i$ and $M = \frac{1}{2} \cdot \sum_{i=1}^{t} k_i \cdot (k_i + 1)$.

7 Elliptic Curves Based Variant

We now extend our algorithm from representations in finite fields to representations in elliptic curve groups. Recall we need to embed $\mathbb{Z}_q^+$ into a group defined over $\mathbb{F}_p$ whose operations can be expressed in terms of the functionality of the homomorphic encryption scheme. This means that the range of the representation should be an algebraic group. We have already seen linear algebraic groups (a.k.a. matrix representations) used in this context in work of Alperin-Sherriff and Peikert, thus as it is natural (to anyone who has studied algebraic groups) to consider algebraic varieties. The finite field case discussed in the previous sections corresponds to the genus zero case, thus the next natural extension would be to examine the genus one case (a.k.a. elliptic curves).

The reason for doing this is the value of q from Table 2 compared to the estimated values from Table 1 are far from optimal. This is because we have few possible group orders of $\mathbb{F}^*_{p^{k_i}}$. The standard trick in this context (used for example in the ECM factorization method, the ECPP primality prover, or even indeed in all of elliptic curve cryptography) is to replace the multiplicative group of a finite field by an elliptic curve group.

Just as before we select a coprime factorization $q = \prod_{i=1}^{t} e_i$ (with the e_i not necessarily prime, but pairwise coprime). But now we require that e_i divides the order of an elliptic curve E_i defined over p^{k_i}. Since the group orders of elliptic curves are distributed roughly uniformly within the Hasse interval it is highly likely that there are such elliptic curves. Determining such curves may however be a hard problem for a fixed value of q; a problem which arose previously in cryptography in [3]. However,

since we have some freedom in selecting q in our scheme we can select q and the E_i simultaneously, and hence finding the elliptic curves will not be a problem.

Again, we fix a polynomial representation of $\mathbb{F}_{p^{k_i}}$, i.e. an irreducible polynomial $f_i(x)$ of degree k_i such that $\mathbb{F}_{p^{k_i}} = \mathbb{F}_p[x]/f_i(x)$, and now we let $G_i \in E_i(\mathbb{F}_{p^{k_i}})$ denote a fixed point on the elliptic curve of order e_i. We now can translate our method into this new setting. For example Equation (1) translates to

$$\mathsf{rep} : \begin{cases} \mathbb{Z}_q^+ \longrightarrow \mathbb{G} = \prod_{i=1}^t E_i(\mathbb{F}_{p^{k_i}}) \\ a \longmapsto ([a_1]G_1, \ldots, [a_t]G_t) \end{cases} \tag{2}$$

where $a_i = a \pmod{e_i}$.

Homomorphic calculations in $\mathbb{G}$ are then performed using Jacobian Projective coordinates. This means that general point addition can be performed with multiplicative depth five and $M' = 16 \cdot M$ homomorphic multiplications. Our method then proceeds as before, except we replace homomorphic multiplication in $\mathbb{F}^*_{p^{k_i}}$ with Jacobian projective point addition in $E_i(\mathbb{F}_{p^{k_i}})$.

The computation of red is then performed as follows. We first homomorphically map the projective points in $\mathbb{G}$ into an affine point. Each such conversion, in component i, requires an $\mathbb{F}_{p^{k_i}}$-field inversion and three $\mathbb{F}_{p^{k_i}}$-field multiplications. If we let DInv_i (resp. MInv_i) denote the depth (resp. number of multiplications in $\mathbb{F}_p$) of the circuit to invert in the field $\mathbb{F}_{p^{k_i}}$. This implies that the conversion of a set of projective points in $\mathbb{G}$ to a set of affine points requires depth $3 + \max_{i=1}^t \mathsf{DInv}_i$ and $4 \cdot M + \sum_{i=1}^t \mathsf{MInv}_i$ homomorphic multiplications over $\mathbb{F}_p$.

Given this final conversion to affine form, we have effectively $E' = E + t$, as opposed to E, variables defining the elements in $\mathbb{G}$. The extra t variables coming from the y-coordinate; it is clear we only need to store t such variables as opposed to E such variables as each x coordinate corresponds to at most two y-coordinates and hence a naive form of homomorphic point compression can be applied.

This means the map red (after the conversion to affine coordinates so as to reduce the multiplicative complexity of the interpolated polynomial) can be expressed as a degree d' map; where we expect d' to be the smallest d' such that ${}^{E'+d'}C_{d'} > q$, which means we expect $d' \approx E' \cdot (2^{\log(q)/\log(E')} - 1)$. This means, as before, that the resulting depth will be $\lceil \log_2 d' \rceil$ and the number of multiplications will be $D(E', d')$.

So putting all of the sub-procedures for bootstrapping together, we find that we can use the elliptic curve variant of our bootstrapping method to bootstrap $\ell^{(R)}/n$ ciphertexts in parallel using a procedure of depth of

$$\lceil \log_2 d' \rceil + 5 \cdot \left(\frac{1}{2} + \lceil \log_2 \ell^{(R)} \rceil + \cdot \lceil \log_2 p \rceil + \cdot \lceil \log_2 \lceil \log_2 p \rceil \rceil + \lceil \log_2 \lambda \rceil \right) + 3 + \max_{i=1}^{t} \mathsf{DInv}_i + O(\log_2 \log_2 \ell^{(R)})$$

and

$$D(E', d') + \lambda \cdot \left(M' + \ell^{(R)} \cdot (M' + 3 \cdot E + 2 \cdot M' \cdot \lceil \log_2 p \rceil) \right) + 4 \cdot M + \sum_{i=1}^{t} \mathsf{MInv}_i + O(\ell^{(R)})$$

multiplications, where $d' \approx \log q / \log E'$, $E' = \sum_{i=1}^{t}(k_i + 1)$, $M = \sum_{i=1}^{t} k_i \cdot (k_i + 1)/2$ and $M' = 16 \cdot M$. Note the $3 \cdot E$ term comes from needing to rotate the three projective coordinates.

However, the ability to use arbitrary q comes at a penalty; the depth required has dramatically increased due to the elliptic curve group operations. For example if we consider a prime p of size roughly 2^{16} and $k = 2$, then we need about 200 levels, as opposed to 56 with the finite field variant. This then strongly influences the required value of N, pushing it up from around $85,000$ to $220,000$. Thus in practice the elliptic curve variant is unlikely to be viable.

8 Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant EP/I03126X, by the European Commission under the H2020 project HEAT and by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079[1].

References

1. J. Alperin-Sheriff and C. Peikert. Practical bootstrapping in quasilinear time. In *CRYPTO*, volume 8042 of *Lecture Notes in Computer Science*, pages 1–20, 2013.
2. J. Alperin-Sheriff and C. Peikert. Faster bootstrapping with polynomial error. In *CRYPTO*, volume 8616 of *Lecture Notes in Computer Science*, pages 297–314, 2014.
3. D. Boneh and R.J. Lipton. Algorithms for black-box fields and their application to cryptography (extended abstract). In *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 283–297, 1996.
4. Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886, 2012.
5. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, pages 309–325. ACM, 2012.
6. Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS*, pages 97–106. IEEE, 2011.
7. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524, 2011.
8. Z. Brakerski and V. Vaikuntanathan. Lattice-based FHE as secure as PKE. In *ITCS*, pages 1–12, 2014.
9. Y. Chen and P.Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20, 2011.

[1] The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

10. Jung Hee Cheon, Jean-Sébastien Coron, Jinsu Kim, Moon Sung Lee, Tancrède Lepoint, Mehdi Tibouchi, and Aaram Yun. Batch fully homomorphic encryption over the integers. In *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 315–335, 2013.
11. A. Choudhury, J. Loftus, E. Orsini, A. Patra, and N.P. Smart. Between a rock and a hard place: Interpolating between MPC and FHE. In *ASIACRYPT*, volume 8270 of *Lecture Notes in Computer Science*, pages 221–240, 2013.
12. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits. In *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, 2013.
13. I. Damgård, V. Pastro, N.P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, 2012.
14. C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.
15. C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178. ACM, 2009.
16. C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148, 2011.
17. C. Gentry, S. Halevi, C. Peikert, and N.P. Smart. Field switching in BGV-style homomorphic encryption. *Journal of Computer Security*, 21(5):663–684, 2013.
18. C. Gentry, S. Halevi, and N. P. Smart. Better bootstrapping in fully homomorphic encryption. In *Public Key Cryptography*, volume 7293 of *Lecture Notes in Computer Science*, pages 1–16, 2012.
19. C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482, 2012.
20. C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867, 2012.
21. C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92, 2013.
22. S. Halevi and V. Shoup. Algorithms in HElib. Cryptology ePrint Archive, Report 2014/106, 2014.
23. T. Lepoint and M. Naehrig. A comparison of the homomorphic encryption schemes FV and YASHE. In *AFRICACRYPT*, volume 8469 of *Lecture Notes in Computer Science*, pages 318–335, 2014.
24. R. Lindner and C. Peikert. Better key sizes (and attacks) for LWE-based encryption. In *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339, 2011.
25. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, 2010.
26. V. Lyubashevsky, C. Peikert, and O. Regev. A toolkit for ring-lwe cryptography. In *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 35–54, 2013.
27. D. Micciancio and O. Regev. Lattice-based cryptography. In *Post-quantum cryptography*, pages 147–191. Springer, 2009.
28. K. Rohloff and D.B. Cousins. A scalable implementation of fully homomorphic encryption built on NTRU. In *Finanical Cryptography*, volume 8438 of *Lecture Notes in Computer Science*, pages 221–234, 2014.
29. N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *PKC*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443, 2010.

30. N.P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography*, 71:57–81, 2014.
31. J. van de Pol and N.P. Smart. Estimating key sizes for high dimensional lattice-based systems. In *IMA Int. Conf.*, volume 8308 of *Lecture Notes in Computer Science*, pages 290–303, 2013.
32. Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43, 2010.

A Parameter Calculation

In [20] a concrete set of parameters for the BGV SHE scheme was given for the case of binary message spaces, and arbitrary L. In [12] this was adapted to the case of message space R_p for 2-power cyclotomic rings, but only for the schemes which could support one level of multiplication gates (i.e. for $L = 1$). In [11] these two approaches were combined, for arbitrary L and p, and the analysis was (slightly) modified to remove the need for a modulus switching upon encryption. In this section we modify again the analysis of [11] to present an analysis which includes a step of field switching from [17]. We assume in this section that the reader is familiar with the analysis and algorithms from [20,11,17].

Our analysis will make extensive use of the following fact: If $a \in R$ be chosen from a distribution such that the coefficients are distributed with mean zero and standard deviation σ, then if ζ_m is a primitive mth root of unity, we can use $6 \cdot \sigma$ to bound $a(\zeta_m)$ and hence the canonical embedding norm of a. If we have two elements with variances σ_1^2 and σ_2^2, then we can bound the canonical norm of their product with $16 \cdot \sigma_1 \cdot \sigma_2$.

Ensuring We Can Evaluate the Required Depth: Recall we have two rings R and S of degree N and n respectively. The ring S is a subring of R and hence n divides N. We require a chain of moduli $q_0 < q_1 \ldots < q_L$ corresponding to each level of the scheme. We assume (for sake of simplicity) that $q_i/q_{i-1} = p_i$ are primes. Thus $q_L = q_0 \cdot \prod_{i=1}^{i=L} p_i$. Also note, that as in [11], we apply a $\mathsf{SHE.LowerLevel}$ (a.k.a. modulus switch) algorithm *before* a multiplication operation. This often leads to lower noise values in practice (which a practical instantiation can make use of). In addition it eliminates the need to perform a modulus switch after encryption, which happened in [20].

We utilize the following constants described in [12], which are worked out for the case of message space defined modulo p (the constants in [12] make use of an additional parameter, arising from the key generation procedure. In our case we can take this constant equal to one). In the following h is the Hamming weight of the secret keys $\mathfrak{sk}^{(R)}$ and $\mathfrak{sk}^{(S)}$.

$$B_{\mathsf{Clean}} = N \cdot p/2 + p \cdot \sigma \cdot \left(\frac{16 \cdot N}{\sqrt{2}} + 6 \cdot \sqrt{N} + 16 \cdot \sqrt{h \cdot N} \right)$$

$$B_{\mathsf{Scale}}^{(R)} = p \cdot \sqrt{3 \cdot N} \cdot \left(1 + \frac{8}{3} \cdot \sqrt{h} \right)$$

$$B^{(S)}_{\mathsf{Scale}} = p \cdot \sqrt{3 \cdot n} \cdot \left(1 + \frac{8}{3} \cdot \sqrt{h}\right)$$
$$B^{(R)}_{\mathsf{Ks}} = p \cdot \sigma \cdot N \cdot \left(1.49 \cdot \sqrt{h \cdot N} + 2.11 \cdot h + 5.54 \cdot \sqrt{h} + 1.96\sqrt{N} + 4.62\right)$$
$$B^{(S)}_{\mathsf{Ks}} = p \cdot \sigma \cdot n \cdot \left(1.49 \cdot \sqrt{h \cdot n} + 2.11 \cdot h + 5.54 \cdot \sqrt{h} + 1.96\sqrt{n} + 4.62\right)$$

As in [20] we define a small "wiggle room" ξ which we set to be equal to eight; this is set to enable a number of additions to be performed without needing to individually account for them in our analysis. These constants arise in the following way:

- A freshly encrypted ciphertext at level L has noise bounded by B_{Clean}.
- In the worst case, when applying $\mathsf{SHE.LowerLevel}$ to a (big ring) ciphertext at level $\mathfrak{l} > L_2 + 1$ with noise bounded by B' one obtains a new ciphertext at level $\mathfrak{l} - 1$ with noise bounded by
$$\frac{B'}{p_{\mathfrak{l}}} + B^{(R)}_{\mathsf{Scale}}.$$
- In the worst case, when applying $\mathsf{SHE.LowerLevel}$ to a (small ring) ciphertext at level $\mathfrak{l} \leq L_2 + 1$ with noise bounded by B' one obtains a new ciphertext at level $\mathfrak{l} - 1$ with noise bounded by
$$\frac{B'}{p_{\mathfrak{l}}} + B^{(S)}_{\mathsf{Scale}}.$$
- When applying the tensor product multiplication operation to (big ring) ciphertexts of a given level $\mathfrak{l} > L_2 + 1$ of noise B_1 and B_2 one obtains a new ciphertext with noise given by
$$B_1 \cdot B_2 + \frac{B^{(R)}_{\mathsf{Ks}} \cdot q_{\mathfrak{l}}}{P_R} + B^{(R)}_{\mathsf{Scale}},$$
where P_R is a value to be determined later.
- When applying the tensor product multiplication operation to (small ring) ciphertexts of a given level $\mathfrak{l} \leq L_2$ of noise B_1 and B_2 one obtains a new ciphertext with noise given by
$$B_1 \cdot B_2 + \frac{B^{(S)}_{\mathsf{Ks}} \cdot q_{\mathfrak{l}}}{P_S} + B^{(S)}_{\mathsf{Scale}},$$
where again P_S is a value to be determined later.

A general evaluation procedure begins with a freshly encrypted ciphertext at level L with noise B_{Clean}. When entering the first multiplication operation we first apply a $\mathsf{SHE.LowerLevel}$ operation to reduce the noise to a universal bounds. $B^{(R)}$, whose value will be determined later. We therefore require

$$\frac{\xi \cdot B_{\mathsf{Clean}}}{p_L} + B^{(R)}_{\mathsf{Scale}} \leq B^{(R)},$$

i.e.

$$p_L \geq \frac{8 \cdot B_{\mathsf{Clean}}}{B^{(R)} - B^{(R)}_{\mathsf{Scale}}}. \tag{3}$$

We now turn to dealing with the SHE.LowerLevel operations which occurs before a multiplication gate at level $\mathfrak{l} \in \{1, \ldots, L-1\} \setminus \{L_2+1\}$. In what follows we assume $\mathfrak{l} > L_2 + 1$, to obtain the equations for $\mathfrak{l} \leq L_2$ one simply replaces the R-constants by their equivalent S-constants. We perform a worst case analysis and assume that the input ciphertexts are at level $\mathfrak{l}$. We can then assume that the input to the tensoring operation in the previous multiplication gate (just after the previous SHE.LowerLevel) was bounded by $B^{(R)}$, and so the output noise from the previous multiplication gate for each input ciphertext is bounded by $(B^{(R)})^2 + B^{(R)}_{\mathsf{Ks}} \cdot q_{\mathfrak{l}}/P_R + B^{(R)}_{\mathsf{Scale}}$. This means the noise on entering the SHE.LowerLevel operation is bounded by ξ times this value, and so to maintain our invariant we require

$$\frac{\xi \cdot (B^{(R)})^2 + \xi \cdot B^{(R)}_{\mathsf{Scale}}}{p_{\mathfrak{l}}} + \frac{\xi \cdot B^{(R)}_{\mathsf{Ks}} \cdot q_{\mathfrak{l}}}{P_R \cdot p_{\mathfrak{l}}} + B^{(R)}_{\mathsf{Scale}} \leq B^{(R)}.$$

Rearranging this into a quadratic equation in $B^{(R)}$ we have

$$\frac{\xi}{p_{\mathfrak{l}}} \cdot (B^{(R)})^2 - B^{(R)} + \left(\frac{\xi \cdot B^{(R)}_{\mathsf{Scale}}}{p_{\mathfrak{l}}} + \frac{\xi \cdot B^{(R)}_{\mathsf{Ks}} \cdot q_{\mathfrak{l}-1}}{P_R} + B^{(R)}_{\mathsf{Scale}} \right) \leq 0.$$

We denote the constant term in this equation by $R_{\mathfrak{l}-1}$. We now assume that all primes $p_{\mathfrak{l}}$ are of roughly the same size (for the ring R), and noting the we need to only satisfy the inequality for the largest modulus $\mathfrak{l} = L-1$ (resp. $\mathfrak{l} = L_2$ for the ring S). We now fix R_{L-2} by trying to ensure that R_{L-2} is close to $B^{(R)}_{\mathsf{Scale}} \cdot (1 + \xi/p_{L-1}) \approx B^{(R)}_{\mathsf{Scale}}$, so we set $R_{L-2} = (1 - 2^{-3}) \cdot B^{(R)}_{\mathsf{Scale}} \cdot (1 + \xi/p_{L-1})$, and obtain

$$P_R \approx 8 \cdot \frac{\xi \cdot B^{(R)}_{\mathsf{Ks}} \cdot q_{L-2}}{B^{(R)}_{\mathsf{Scale}}}, \tag{4}$$

since $B^{(R)}_{\mathsf{Scale}} \cdot (1 + \xi/p_{L-1}) \approx B^{(R)}_{\mathsf{Scale}}$. Similarly for the small ring we find

$$P_S \approx 8 \cdot \frac{\xi \cdot B^{(S)}_{\mathsf{Ks}} \cdot q_{L_2-1}}{B^{(S)}_{\mathsf{Scale}}}, \tag{5}$$

To ensure we have a solution we require $1 - 4 \cdot \xi \cdot R_{L-2}/p_{L-1} \geq 0$, (resp. $1 - 4 \cdot \xi \cdot R_{L_2-1}/p_{L_2} \geq 0$) which implies we should take, for $i = 2, \ldots, L-1$,

$$p_i \approx \begin{cases} 4 \cdot \xi \cdot R_{L-2} \approx 32 \cdot B^{(R)}_{\mathsf{Scale}} = p_R & \text{For } i = L_2 + 2, \ldots, L-1, \\ 4 \cdot \xi \cdot R_{L_2-1} \approx 32 \cdot B^{(S)}_{\mathsf{Scale}} = p_S & \text{For } i = 1, \ldots, L_2. \end{cases} \tag{6}$$

We now examine what happens at level L_2+1 when we perform a ring switch operation. Following Lemma 3.2 of [17] we know the noise increases by a factor of $(p/2) \cdot \sqrt{N/n}$. The noise output from the previous multiplication gate is bounded by $(B^{(R)})^2 + B^{(R)}_{\mathsf{Ks}} \cdot$

$q_{L_2+2}/P_R + B^{(R)}_{\mathsf{Scale}}$. Note that

$$\frac{B^{(R)}_{\mathsf{Ks}} \cdot q_{L_2+2}}{P_R} \approx \frac{B^{(R)}_{\mathsf{Ks}} \cdot q_{L_2+2} \cdot B^{(R)}_{\mathsf{Scale}}}{8 \cdot \xi \cdot B^{(R)}_{\mathsf{Ks}} \cdot q_{L-2}} \approx \frac{B^{(R)}_{\mathsf{Scale}}}{8 \cdot \xi \cdot p_R^{L_1-4}}$$

Thus the we know that the noise after the ring switch operation is bounded by

$$B_{\mathsf{RingSwitch}} = \frac{p}{2} \cdot \sqrt{N/n} \cdot \left((B^{(R)})^2 + \frac{B^{(R)}_{\mathsf{Scale}}}{8 \cdot \xi \cdot p_R^{L_1-4}} + B^{(R)}_{\mathsf{Scale}} \right).$$

We now modulus switch down to level L_2, and obtain a ciphertext (over the ring S) with noise bounded by

$$\frac{B_{\mathsf{RingSwitch}}}{p_{L_2+1}} + B^{(S)}_{\mathsf{Scale}}.$$

We would like this to be less than the universal bound $B^{(S)}$, which implies

$$p_{L_2+1} \geq \frac{B_{\mathsf{RingSwitch}}}{B^{(S)} - B^{(S)}_{\mathsf{Scale}}}. \tag{7}$$

We now need to estimate the size of p_0. Due to the above choices the ciphertext to which we apply the bootstrapping has norm bound by $B^{(S)}$. This means that we require

$$q_0 = p_0 \geq 2 \cdot B^{(S)} \cdot c_{m'}, \tag{8}$$

to ensure a valid decryption/bootstrapping procedure. Recall $c_{m'}$ is the ring constant for the polynomial ring S and it depends only on m' (see [13] for details).

Ensuring We Have Security: The works before [31,23], such as Lindner and Peikert [24], did not include the rank of the lattice into account when estimating the cost of the attacker. The reason is that the lattice rank appears to be only a second order term in the cost of the attack. However, for applications such as FHE, the dimension is usually very big, e.g. 2^{16}, and lattice algorithms are often polynomial in the rank. Therefore, even as a second order term it can contribute significantly to the cost of the attack. The largest modulus used in our big ring (resp. small ring) key switching matrices, i.e. the largest modulus used in an LWE instance, is given by $Q_{L-1} = P_R \cdot q_{L-1}$ (resp. $Q_{L_2} = P_S \cdot q_{L_2}$).

We recall the approach of [31,23] here. First, fix some security level as measured in enumeration nodes, e.g. 2^{128}. Now, use estimates by Chen and Nguyen [9] are used to determine the cost of running BKZ 2.0 for various block sizes β. Combining this with the security level gives an upper bound on the rounds an attacker can perform, depending on β. Then, for various lattice dimensions r, the BKZ 2.0 simulator by Chen and Nguyen is used to determine the quality of the vector as measured by the root-Hermite factor $\delta(\beta, r) = (\|\mathbf{b}\|/\mathrm{vol}(L)^{1/r})^{1/r}$. Now, the best possible root-Hermite factor achievable by the attacker is given by $\delta(r) = \min_\beta \delta(\beta, r)$

In LWE, the relevant parameters for the security are the ring dimension n (resp. N), the modulus $Q = Q_{L_2}$ (resp. $Q = Q_{L-1}$) and the standard deviation σ. Note that in most scenarios, an adversary can choose how many LWE samples he uses in his attack. This number r is equal to the rank of the lattice. The distinguishing attack against LWE uses a short vector in the dual SIS lattice to distinguish the LWE distribution from the uniform distribution. More precisely, an adversary can distinguish between these two distributions with distinguishing advantage ε if the shortest vector he can obtain (in terms of its root-Hermite factor) satisfies

$$\delta(r)^r \cdot Q^{n/r-1} \cdot \sigma < \sqrt{-\log(\varepsilon)/\pi}.$$

It follows that in order for our system to be secure against the previously described adversary, we need that

$$\log_2(Q) \leq \min_{r>n} \frac{r^2 \cdot \log_2(\delta(r)) + r \cdot \log_2(\sigma/\alpha)}{r-n}, \tag{9}$$

where $\alpha = \sqrt{-\log(\varepsilon)/\pi}$. See also[27,24,23] for more information. For every n we can now compute an upper bound on $\log_2(q)$ by iterating the right hand side of Equation (9) over m and selecting the minimum.

Putting it all together As in [20,12], we set $\sigma = 3.2$, $B^{(R)} = 2 \cdot B^{(R)}_{\mathsf{Scale}}$ and $B^{(S)} = 2 \cdot B^{(S)}_{\mathsf{Scale}}$. From our equations (3), (4), (5), (6), (7), and (8) we obtain equations for p_i for $i = 0, \ldots, L$, P_R and P_S in terms of n, N, L, h and the security level κ.

B Example Parameters

In Appendix A we present a calculation of suitable parameters for our scheme, and the resulting complexity of the polynomial representation of red, here we work out a concrete set of parameters for various plaintext moduli p.

We target $\kappa = 128$-bits of security, and set the Hamming weight h of the secret key $\mathfrak{sk}$ to be 64 as in [20,12]. On input N and n the to the formulae in Appendix A we obtain an upper bounds on $\log(Q_{L-1})$ and $\log(Q_{L_2})$. We now use equations (3)-(8) from the Appendix for different values of the plaintext modulus p to obtain a lower bound on $\log(Q_{L-1})$ and $\log(Q_{L_2})$. Then, we increase N and n until the lower bound on Q_{L-1} and Q_{L_2} from the functionality is below the upper bound from the security analysis. In this way we obtain lower bounds for N and n.

In Table 1 we consider four different values of p; for simplicity we also set $t = 1$ in (1), i.e. $\mathbb{G} = \mathbb{F}^*_{p^k}$, for a suitable choice of k. After finding approximate values for N, n and q we can then search for exact values of N, n and q. More precisely, we are looking for cyclotomic rings R and S such that the degree $N = \phi(m)$ of $F(X) = \Phi_m(X)$ and $n = \phi(m')$ of $f(x) = \Phi_{m'}(X)$ are larger than the bounds above and n divides both N and $\ell^{(R)}$ (the number of plaintext slots associated with R). In addition we require that q divides $p^k - 1$. See Table 2 for some values.

Notice that the value of q is strongly influenced by the ring constant $c_{m'}$. In Table 1 we set $c_{m'} = 1.28$ (i.e. we assume the best case of m' being prime), whereas in Table 2

Table 1. Lower bounds on N and n

p	κ	$c = \ell^{(R)}/n$	$n \approx$	$N \approx$	$q \approx$
2	128	1	860	23100	11637
		2		24100	
$\approx 2^8$	128	1	1040	51800	1635087
		2		53100	
		$3, 4,$		56000	
		$[5, \ldots, 10]$		57600	
$\approx 2^{16}$	128	1	1300	96000	467989106
		$2, 3$		98500	
		$[4, \ldots, 10]$		103000	
$\approx 2^{32}$	128	1	1750	181000	$3.558467651 \cdot 10^{13}$
		2		183000	
		$[3, \ldots, 10]$		185000	

Table 2. A concrete set of cyclotomic rings with an estimation of the number of multiplications and the depth required to perform our bootstrapping step

p	m	$N = \phi(m)$	m'	$n = \phi(m')$	$c_{m'}$	$\ell^{(R)}/n$	k	L	# Mults	q
2	31775	24000	1271	1200	3.93	1	16	23	$\approx 8.3 \cdot 10^6$	65535
	32767	27000	1057	900	2.69	2	15	23	$\approx 1.02 \cdot 10^7$	32767
$2^8 + 1$	62419	51840	1687	1440	2.72	1	3	40	$\approx 4.6 \cdot 10^6$	4243648
	91149	58080	1321	1320	1.28	1	3	39	$\approx 2.3 \cdot 10^6$	2121824
	137384	63360	1321	1320	1.28	4	3	41	$\approx 3.5 \cdot 10^6$	2121824
$2^{16} + 1$	113993	100800	2651	2400	2.9	1	2	56	$\approx 1.5 \cdot 10^9$	2147549184
	160977	102608	2333	2332	1.28	2	2	58	$\approx 6.3 \cdot 10^8$	715849728
	272200	108800	1361	1360	1.28	4	2	57	$\approx 4.8 \cdot 10^8$	536887296
$2^{32} + 15$	198203	183040	2227	2080	3.6	1	2	79	$\approx 1.1 \cdot 10^{14}$	414161297767368
	202051	199872	2083	2082	1.28	4	2	79	$\approx 3.9 \cdot 10^{13}$	50637664608480
	352317	190512	2649	1764	1.81	6	2	82	$\approx 5.1 \cdot 10^{14}$	50637664608480

we compute the actual value of the ring constant for each cyclotomic ring we consider. For example for $p = 2$, in Table 1 we obtain an approximate value $q \approx 11637$, but in Table 2 we need a larger value due to the additional condition that q divides $p^k - 1$, and the ring constant, which is bigger than 1.27 for $m' = 1271$ and $m' = 1057$.

Multiparty Computation from Somewhat Homomorphic Encryption

Ivan Damgård[1], Valerio Pastro[1], Nigel Smart[2], and Sarah Zakarias[1]

[1] Department of Computer Science, Aarhus University
[2] Department of Computer Science, Bristol University

Abstract. We propose a general multiparty computation protocol secure against an active adversary corrupting up to $n-1$ of the n players. The protocol may be used to compute securely arithmetic circuits over any finite field $\mathbb{F}_{p^k}$. Our protocol consists of a preprocessing phase that is both independent of the function to be computed and of the inputs, and a much more efficient online phase where the actual computation takes place. The online phase is unconditionally secure and has total computational (and communication) complexity linear in n, the number of players, where earlier work was quadratic in n. Moreover, the work done by each player is only a small constant factor larger than what one would need to compute the circuit in the clear. We show this is optimal for computation in large fields. In practice, for 3 players, a secure 64-bit multiplication can be done in 0.05 ms. Our preprocessing is based on a somewhat homomorphic cryptosystem. We extend a scheme by Brakerski et al., so that we can perform distributed decryption and handle many values in parallel in one ciphertext. The computational complexity of our preprocessing phase is dominated by the public-key operations, we need $O(n^2/s)$ operations per secure multiplication where s is a parameter that increases with the security parameter of the cryptosystem. Earlier work in this model needed $\Omega(n^2)$ operations. In practice, the preprocessing prepares a secure 64-bit multiplication for 3 players in about 13 ms.

1 Introduction

A central problem in theoretical cryptography is that of secure multiparty computation (MPC). In this problem n parties, holding private inputs $x_1, \ldots, x_n$, wish to compute a given function $f(x_1, \ldots, x_n)$. A protocol for doing this securely should be such that honest players get the correct result and this result is the only new information released, even if some subset of the players is controlled by an adversary.

In the case of *dishonest majority*, where more than half the players are corrupt, unconditionally secure protocols cannot exist. Under computational assumptions, it was shown in [8] how to construct UC-secure MPC protocols that handle the case where all but one of the parties are actively corrupted. The public-key machinery one needs for this is typically expensive so efficient solutions are hard to design for dishonest majority. Recently, however, a new approach has been proposed making such protocols more practical. This approach works as follows: one first designs a general MPC protocol in the *preprocessing model*, where access to a "trusted dealer" is assumed. The dealer does not need to know the function to be computed, nor the inputs, he just supplies raw material for the computation before it starts. This allows the "online" protocol to use only cheap information theoretic primitives and hence be efficient. Finally, one implements the trusted dealer by a secure protocol using public-key techniques, this protocol can then be run in a preprocessing phase. The current state of the art in this respect are the protocols in Bendlin et al., Damgård/Orlandi and Nielsen et al. [5, 13, 25]. The "MPC-in-the-head" technique of Ishai et al. [18, 17] has similar overall asymptotic complexity, but larger constants and a less efficient online phase.

Recently, another approach has become possible with the advent of Fully Homomorphic Encryption (FHE) by Gentry [15]. In this approach all parties first encrypt their input under the

FHE scheme; then they evaluate the desired function on the ciphertexts using the homomorphic properties, and finally they perform a distributed decryption on the final ciphertexts to get the results. The advantage of the FHE-based approach is that interaction is only needed to supply inputs and get output. However, the low bandwidth consumption comes at a price; current FHE schemes are very slow and can only evaluate small circuits, i.e., they actually only provide what is known as somewhat homomorphic encryption (SHE). This can be circumvented in two ways; either by assuming circular security and implementing an expensive bootstrapping operation, or by extending the parameter sizes to enable a "levelled FHE" scheme which can evaluate circuits of large degree (exponential in the number of levels) [6]. The main cost, much like other approaches, is in terms of the number of multiplications in the arithmetic circuit. So whilst theoretically appealing the approach via FHE is not competitive in practice with the traditional MPC approach.

1.1 Contributions of this paper.

Optimal Online Phase. We propose an MPC protocol in the preprocessing model that computes securely an arithmetic circuit C over any finite field $\mathbb{F}_{p^k}$. The protocol is statistically UC-secure against active and adaptive corruption of up to $n-1$ of the n players, and we assume synchronous communication and secure point-to-point channels. Measured in elementary operations in $\mathbb{F}_{p^k}$ the total amount of work done is $O(n \cdot |C| + n^3)$ where $|C|$ is the size of C. All earlier work in this model had complexity $\Omega(n^2 \cdot |C|)$. A similar improvement applies to the communication complexity and the amount of data one needs to store from the preprocessing. Hence, the work done by each player in the online phase is essentially independent of n. Moreover, it is only a small constant factor larger than what one would need to compute the circuit in the clear. This is the first protocol in the preprocessing model with these properties[3].

Finally, we show a lower bound implying that w.r.t the amount of data required from the preprocessing, our protocol is optimal up to a constant factor. We also obtain a similar lower bound on the number of bit operations required, and hence the computational work done in our protocol is optimal up to poly-logarithmic factors.

All results mentioned here hold for the case of large fields, i.e., where the desired error probability is $(1/p^k)^c$, for a small constant c. Note that many applications of MPC need integer arithmetic, modular reductions, conversion to binary, etc., which we can emulate by computing in $\mathbb{F}_p$ with p large enough to avoid overflow. This naturally leads to computing with large fields. As mentioned, our protocol works for all fields, but like earlier work in this model it is less efficient for small fields by a factor of essentially $\lceil \frac{\mathsf{sec}}{\log p^k} \rceil$ for error probability $2^{-\Theta(\mathsf{sec})}$, see Appendix A.4 for details.

Obtaining our result requires new ideas compared to [5], which was previously state of the art and was based on additive secret sharing where each share in a secret is authenticated using an information theoretic Message Authentication Code (MAC). Since each player needs to have his own key, each of the n shares need to be authenticated with n MACs, so this approach is inherently quadratic in n. Our idea is to authenticate the secret value itself instead of the shares, using a single global key. This seems to lead to a "chicken and egg" problem since one cannot check a MAC without knowing the key, but if the key is known, MACs can be forged. Our solution to this

[3] With dishonest majority, successful termination cannot be guaranteed, so our protocols simply abort if cheating is detected. We do not, however, identify *who* cheated, indeed the standard definition of secure function evaluation does not require this. Identification of cheaters is possible but we do not know how to do this while maintaining complexity linear in n.

involves secret sharing the key as well, carefully timing when values are revealed, and various tricks to reduce the amortized cost of checking a set of MACs.

Efficient use of FHE for MPC. As a conceptual contribution we propose what we believe is "the right" way to use FHE/SHE for *computationally* efficient MPC, namely to use it for implementing a preprocessing phase. The observation is that since such preprocessing is typically based on the classic circuit randomization technique of Beaver [3], it can be done by evaluating in parallel many small circuits of small multiplicative depth (in fact depth 1 in our case). Thus SHE suffices, we do not need bootstrapping, and we can use the SHE SIMD approach of [28] to handle many values in parallel in a single ciphertext.

To capitalize on this idea, we apply the SIMD approach to the cryptosystem from [7] (see also [16] where this technique is also used). To get the best performance, we need to do a non-trivial analysis of the parameter values we can use, and we prove some results on norms of embeddings of a cyclotomic field for this purpose. We also design a distributed decryption procedure for our cryptosystem. This protocol is only robust against passive attacks. Nevertheless, this is sufficient for the overall protocol to be actively secure. Intuitively, this is because the only damage the adversary can do is to add a known error term to the decryption result obtained. The effect of this for the online protocol is that certain shares of secret values may be incorrect, but this will caught by the check involving the MACs. Finally we adapt a zero-knowledge proof of plaintext knowledge from [5] for our purpose and in particular we improve the analysis of the soundness guarantees it offers. This influences the choice of parameters for the cryptosystem and therefore improves overall performance.

An Efficient Preprocessing Protocol. As a result of the above, we obtain a constant-round preprocessing protocol that is UC-secure against active and static corruption of $n-1$ players assuming the underlying cryptosystem is semantically secure, which follows from the polynomial (PLWE) assumption. UC-security for dishonest majority cannot be obtained without a set-up assumption. In this paper we assume that a key pair for our cryptosystem has been generated and the secret key has been shared among the players.

Whereas previous work in the preprocessing/online model [5, 13] use $\Omega(n^2)$ public-key operations per secure multiplication, we only need $O(n^2/s)$ operations, where s is a number that grows with the security parameter of the SHE scheme (we have $s \approx 12000$ in our concrete instantiation for computing in $\mathbb{F}_p$ where $p \approx 2^{64}$). We stress that our adapted scheme is exactly as efficient as the basic version of [7] that does not allow this optimization, so the improvement is indeed "genuine".

In comparison to the approach mentioned above where one uses FHE throughout the protocol, our combined preprocessing and online phase achieves a result that is incomparable from a theoretical point of view, but much more practical: we need more communication and rounds, but the computational overhead is much smaller – we need $O(n^2/s \cdot |C|)$ public key operations compared to $O(n \cdot |C|)$ for the FHE approach, where for realistic values of n and s, we have $n^2/s \ll n$. Furthermore, we only need a low depth SHE which is much more efficient in the first place. And finally, we can push all the work using SHE into a, function independent, preprocessing phase.

Performance in practice. Both the preprocessing and online phase have been implemented and tested for 3 players on up-to-date machines connected on a LAN. The preprocessing takes about 13 ms amortized time to prepare one multiplication in $\mathbb{F}_p$ for a 64-bit p, with security level corresponding roughly to 1024 bit RSA and an error probability of 2^{-40} for the zero-knowledge proofs

(the error probability can be lowered to 2^{-80} by repeating the ZK proofs which will at most double the time). This is 2-3 orders of magnitude faster than preliminary estimates for the most efficient instantiation of [5]. The online phase executes a secure 64-bit multiplication in 0.05 ms amortized time. These rough orders of magnitude, and the ability to deal with a non-trivial number of players, are born out by a recent implementation of the protocols described in this paper [11].

Concurrent Related Work. In recent independent work [24, 2, 16], Meyers at al., Asharov et al. and Gentry et al. also use an FHE scheme for multiparty computation. They follow the pure FHE approach mentioned above, using a threshold decryption protocol tailored to the specific FHE scheme. They focus primarily on round complexity, while we want to minimize the computational overhead. We note that in [16], Gentry et al. obtain small overhead by showing a way to use the FHE SIMD approach for computing any circuit homomorphically. However, this requires full FHE with bootstrapping (to work on arbitrary circuits) and does not (currently) lead to a practical protocol.

In [25], Nielsen et al. consider secure computing for Boolean Circuits. Their online phase is similar to that of [5], while the preprocessing is a clever and very efficient construction based on Oblivious Transfer. This result is complementary to ours in the sense that we target computations over large fields which is good for some applications whereas for other cases, Boolean Circuits are the most compact way to express the desired computation. Of course, one could use the preprocessing from [25] to set up data for our online phase, but current benchmarks indicate that our approach is faster for large fields, say of size 64 bits or more.

We end the introduction by covering some basic notation which will be used throughout this paper. For a vector $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ we denote by $\|\mathbf{x}\|_\infty := \max_{1 \leq i \leq n} |x_i|$, $\|\mathbf{x}\|_1 := \sum_{1 \leq i \leq n} |x_i|$ and $\|\mathbf{x}\|_2 := \sqrt{\sum |x_i|^2}$. We let $\epsilon(\kappa)$ denote an unspecified negligible function of κ. If S is a set we let $x \leftarrow S$ denote assignment to the variable x with respect to a uniform distribution on S; we use $x \leftarrow s$ for a value s as shorthand for $x \leftarrow \{s\}$. If A is an algorithm $x \leftarrow A$ means assign to x the output of A, where the probability distribution is over the random coins of A. Finally $x := y$ means "x is defined to be y".

2 Online Protocol

Our aim is to construct a protocol for arithmetic multiparty computation over $\mathbb{F}_{p^k}$ for some prime p. More precisely, we wish to implement the ideal functionality $\mathcal{F}_{\text{AMPC}}$, presented in Figure 15 in Appendix Ethe full version. Our MPC protocol is structured in a preprocessing (or offline) phase and an online phase. We start out in this section by presenting the online phase which assumes access to an ideal functionality $\mathcal{F}_{\text{PREP}}$ (Figure 16 of Appendix E). In Section 5 we show how to implement this functionality in an independent preprocessing phase.

In our specification of the online protocol, we assume for simplicity that a broadcast channel is available at unit cost, that each party has only one input, and only one public output value is to be computed. In Appendix A.3 we explain how to implement the broadcasts we need from point-to-point channels and lift the restriction on the number of inputs and outputs without this affecting the overall complexity.

Before presenting the concrete online protocol we give the intuition and motivation behind the construction. We will use unconditionally secure MACs to protect secret values from being manipulated by an active adversary. However, rather than authenticating shares of secret values as

in [5], we authenticate the shared value itself. More concretely, we will use a global key α chosen randomly in $\mathbb{F}_{p^k}$, and for each secret value a, we will share a additively among the players, and we also secret-share a MAC αa. This way to represent secret values is linear, just like the representation in [5], and we can therefore do secure multiplication based on multiplication triples à la Beaver [3] that we produce in the preprocessing.

An immediate problem is that opening a value reliably seems to require that we check the MAC, and this requires players know α. However, as soon as α is known, MACs on other values can be forged. We solve this problem by postponing the check on the MACs (of opened values) to the output phase (of course, this may mean that some of the opened values are incorrect). During the output phase players generate a random linear combination of both the opened values and their shares of the corresponding MACs; they commit to the results and only then open α (see Figure 1). The intuition is that, because of the commitments, when α is revealed it is too late for corrupt players to exploit knowledge of the key. Therefore, if the MAC checks out, all opened values were correct with high probability, so we can trust that the output values we computed are correct and can safely open them.

Protocol Π_{Online}

Initialize: The parties first invoke the preprocessing to get the shared secret key $[\![\alpha]\!]$, a sufficient number of multiplication triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, and pairs of random values $\langle r\rangle, [\![r]\!]$, as well as single random values $[\![t]\!], [\![e]\!]$. Then the steps below are performed in sequence according to the structure of the circuit to compute.

Input: To share P_i's input x_i, P_i takes an available pair $\langle r\rangle, [\![r]\!]$. Then, do the following:

1. $[\![r]\!]$ is opened to P_i (if it is known in advance that P_i will provide input, this step can be done already in the preprocessing stage).
2. P_i broadcasts $\epsilon \leftarrow x_i - r$.
3. The parties compute $\langle x_i\rangle \leftarrow \langle r\rangle + \epsilon$.

Add: To add two representations $\langle x\rangle, \langle y\rangle$,the parties locally compute $\langle x\rangle + \langle y\rangle$.

Multiply: To multiply $\langle x\rangle, \langle y\rangle$ the parties do the following:

1. They take two triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle), (\langle f\rangle, \langle g\rangle, \langle h\rangle)$ from the set of the available ones and check that indeed $a \cdot b = c$.
   - Open a representation of a random value $[\![t]\!]$.
   - partially open $t \cdot \langle a\rangle - \langle f\rangle$ to get ρ and $\langle b\rangle - \langle g\rangle$ to get σ
   - evaluate $t \cdot \langle c\rangle - \langle h\rangle - \sigma \cdot \langle f\rangle - \rho \cdot \langle g\rangle - \sigma \cdot \rho$, and partially open the result.
   - If the result is not zero the players abort, otherwise go on with $\langle a\rangle, \langle b\rangle, \langle c\rangle$.

   Note that this check could in fact be done as part of the preprocessing. Moreover, it can be done for all triples in parallel, and so we actually need only one random value t.
2. The parties partially open $\langle x\rangle - \langle a\rangle$ to get ϵ and $\langle y\rangle - \langle b\rangle$ to get δ and compute $\langle z\rangle \leftarrow \langle c\rangle + \epsilon\langle b\rangle + \delta\langle a\rangle + \epsilon\delta$

Output: We enter this stage when the players have $\langle y\rangle$ for the output value y, but this value has been not been opened (the output value is only correct if players have behaved honestly). We then do the following:

1. Let $a_1, \ldots, a_T$ be all values publicly opened so far, where $\langle a_j\rangle = (\delta_j, (a_{j,1}, \ldots, a_{j,n}), (\gamma(a_j)_1, \ldots, \gamma(a_j)_n))$. Now, a random value $[\![e]\!]$ is opened, and players set $e_i = e^i$ for $i = 1, \ldots, T$. All players compute $a \leftarrow \sum_j e_j a_j$.
2. Each P_i calls $\mathcal{F}_{\text{Com}}$ to commit to $\gamma_i \leftarrow \sum_j e_j \gamma(a_j)_i$. For the output value $\langle y\rangle$, P_i also commits to his share y_i, and his share $\gamma(y)_i$ in the corresponding MAC.
3. $[\![\alpha]\!]$ is opened.
4. Each P_i asks $\mathcal{F}_{\text{Com}}$ to open γ_i, and all players check that $\alpha(a + \sum_j e_j\delta_j) = \sum_i \gamma_i$. If this is not OK, the protocol aborts. Otherwise the players conclude that the output value is correctly computed.
5. To get the output value y, the commitments to $y_i, \gamma(y)_i$ are opened. Now, y is defined as $y := \sum_i y_i$ and each player checks that $\alpha(y + \delta) = \sum_i \gamma(y)_i$, if so, y is the output.

Fig. 1. The online phase.

Representation of values and MACs. In the online phase each shared value $a \in \mathbb{F}_{p^k}$ is represented as follows

$$\langle a \rangle := (\delta, (a_1, \ldots, a_n), (\gamma(a)_1, \ldots, \gamma(a)_n))$$

where $a = a_1 + \cdots + a_n$ and $\gamma(a)_1 + \cdots + \gamma(a)_n = \alpha(a + \delta)$. Player P_i holds $a_i, \gamma(a)_i$ and δ is public. The interpretation is that $\gamma(a) \leftarrow \gamma(a)_1 + \cdots + \gamma(a)_n$ is the MAC authenticating a under the global key α.

Computations. Using the natural component-wise addition of representations, and suppressing the underlying choices of $a_i, \gamma(a)_i$ for readability, we clearly have for secret values a, b and public constant e that

$$\langle a \rangle + \langle b \rangle = \langle a + b \rangle \quad e \cdot \langle a \rangle = \langle ea \rangle, \quad \text{and} \quad e + \langle a \rangle = \langle e + a \rangle,$$

where $e + \langle a \rangle := (\delta - e, (a_1 + e, a_2, \ldots, a_n), (\gamma(a)_1, \ldots, \gamma(a)_n))$. This possibility to easily add a public value is the reason for the "public modifier" δ in the definition of $\langle \cdot \rangle$. It is now clear that we can do secure linear computations directly on values represented this way.

What remains is multiplications: here we use the preprocessing. We would like the preprocessing to output random triples $\langle a \rangle, \langle b \rangle, \langle c \rangle$, where $c = ab$. However, our preprocessing produces triples which satisfy $c = ab + \Delta$, where Δ is an error that can be introduced by the adversary. We therefore need to check the triple before we use it. The check can be done by "sacrificing" another triple $\langle f \rangle, \langle g \rangle, \langle h \rangle$, where the same multiplicative equality should hold (see the protocol for details). Given such a valid triple, we can do multiplications in the following standard way: To compute $\langle xy \rangle$ we first open $\langle x \rangle - \langle a \rangle$ to get ϵ, and $\langle y \rangle - \langle b \rangle$ to get δ. Then $xy = (a + \epsilon)(b + \delta) = c + \epsilon b + \delta a + \epsilon\delta$. Thus, the new representation can be computed as

$$\langle x \rangle \cdot \langle y \rangle = \langle c \rangle + \epsilon \langle b \rangle + \delta \langle a \rangle + \epsilon\delta.$$

An important note is that during our protocol we are actually not guaranteed that we are working with the correct results, since we do not immediately check the MACs of the opened values. During the first part of the protocol, parties will only do what we define as a *partial opening*, meaning that for a value $\langle a \rangle$, each party P_i sends a_i to P_1, who computes $a = a_1 + \cdots + a_n$ and broadcasts a to all players. We assume here for simplicity that we always go via P_1, whereas in practice, one would balance the workload over the players.

As sketched earlier we postpone the checking to the end of the protocol in the output phase. To check the MACs we need the global key α. We get α from the preprocessing but in a slightly different representation:

$$[\![\alpha]\!] := ((\alpha_1, \ldots, \alpha_n), (\beta_i, \gamma(\alpha)_1^i, \ldots, \gamma(\alpha)_n^i)_{i=1,\ldots,n})),$$

where $\alpha = \sum_i \alpha_i$ and $\sum_j \gamma(\alpha)_i^j = \alpha\beta_i$. Player P_i holds $\alpha_i, \beta_i, \gamma(\alpha)_1^i, \ldots, \gamma(\alpha)_n^i$. The idea is that $\gamma(\alpha)_i \leftarrow \sum_j \gamma(\alpha)_i^j$ is the MAC authenticating α under P_i's private key β_i. To open $[\![\alpha]\!]$ each P_j sends to each P_i his share α_j of α and his share $\gamma(\alpha)_i^j$ of the MAC on α made with P_i's private key and then P_i checks that $\sum_j \gamma(\alpha)_i^j = \alpha\beta_i$. (To open the value to only one party P_i, the other parties will simply send their shares only to P_i, who will do the checking. Only shares of α and $\alpha\beta_i$ are needed.)

Finally, the preprocessing will also output n pairs of a random value r in both of the presented representations $\langle r \rangle, [\![r]\!]$. These pairs are used in the Input phase of the protocol.

The full protocol for the online phase is shown in Figure 1. It assumes access to a commitment functionality $\mathcal{F}_{\text{COM}}$ that simply receives values to commit to from players, stores them and reveals a value to all players on request from the committer. Such a functionality could be implemented efficiently based, e.g., on Paillier encryption or the DDH assumption [12, 19]. However, we show in Appendix A.3 that we can do ideal commitments based only on $\mathcal{F}_{\text{PREP}}$ and with cost $O(n^2)$ computation and communication.

Complexity. The (amortized) cost of a secure multiplication is easily seen to be $O(n)$ local elementary operations in $\mathbb{F}_{p^k}$, and communication of $O(n)$ field elements. Linear operations have the same computational cost but require no communication. The input stage requires $O(n)$ communication and computation to open $[\![r]\!]$ to P_i and one broadcast. Doing the output stage requires opening $O(n)$ commitments. In fact, the total number of commitments used is also $O(n)$, so this adds an $O(n^3)$ term to the complexity. In total, we therefore get the complexity claimed in the introduction: $O(n \cdot |C| + n^3)$ elementary field operations and storage/communication complexity $O(n \cdot |C| + n^3)$ field elements.

We can now state the theorem on security of the online phase, and its proof is in Appendix A.3.

Theorem 1. *In the $\mathcal{F}_{\text{PREP}}, \mathcal{F}_{\text{COM}}$-hybrid model, the protocol Π_{ONLINE} implements $\mathcal{F}_{\text{AMPC}}$ with statistical security against any static[4] active adversary corrupting up to $n-1$ parties.*

Based on a result from [29], we can also show a lower bound on the amount of preprocessing data and work required for a protocol. The proof is in Appendix B.

Theorem 2. *Assume a protocol π is the preprocessing model can compute any circuit over $\mathbb{F}_{p^k}$ of size at most S, with security against active corruption of at most $n-1$ players. We assume that the players supply roughly the same number of inputs ($O(S/n)$ each), and that any any player may receive output. Then the preprocessing must output $\Omega(S \log p^k)$ bits to each player, and for any player P_i, there exists a circuit C satisfying the conditions above, where secure computation of C requires P_i to execute an expected number of bit operations that is $\Omega(S \log p^k)$.*

It is easy to see that our protocol satisfies the conditions in the the theorem and that it meets the first bound up to a constant factor and the second up to a poly-logarithmic factor (as a function of the security parameter).

3 The Abstract Somewhat Homomorphic Encryption Scheme

In this section we specify the abstract properties we need for our cryptosystem. A concrete instantiation is found in Section 6.

We first define the plaintext space M. This will be given by a direct product of finite fields $(\mathbb{F}_{p^k})^s$ of characteristic p. Componentwise addition and multiplication of elements in M will be denoted by $+$ and $\cdot$. We assume there is an injective encoding function encode which takes elements in $(\mathbb{F}_{p^k})^s$ to elements in a ring R which is equal $\mathbb{Z}^N$ (as a $\mathbb{Z}$-module) for some integer N. We also assume a decode function which takes *arbitrary* elements in $\mathbb{Z}^N$ and returns an element in $(\mathbb{F}_{p^k})^s$. We require that for all $\mathbf{m} \in M$ that $\mathsf{decode}(\mathsf{encode}(\mathbf{m})) = \mathbf{m}$ and that the decode operation is compatible with the characteristic of the field, i.e. for any $\mathbf{x} \in \mathbb{Z}^N$ we have $\mathsf{decode}(\mathbf{x}) = \mathsf{decode}(\mathbf{x}$

[4] The protocol is in fact adaptively secure, here we only show static security since our preprocessing is anyway only statically secure.

(mod p)). And finally that the encoding function produces "short" vectors. More precisely, that for all $\mathbf{m} \in (\mathbb{F}_{p^k})^s$ $\|\mathsf{encode}(\mathbf{m})\|_\infty \leq \tau$ where $\tau = p/2$.

The two operations in R will be denoted by $+$ and $\cdot$. The addition operation in R is assumed to be componentwise addition, whereas we make no assumption on multiplication. All we require is that the following properties hold, for all elements $\mathbf{m}_1, \mathbf{m}_2 \in M$;

$$\mathsf{decode}(\mathsf{encode}(\mathbf{m}_1) + \mathsf{encode}(\mathbf{m}_2)) = \mathbf{m}_1 + \mathbf{m}_2,$$
$$\mathsf{decode}(\mathsf{encode}(\mathbf{m}_1) \cdot \mathsf{encode}(\mathbf{m}_2)) = \mathbf{m}_1 \cdot \mathbf{m}_2.$$

From now on, when we discuss the plaintext space M we assume it comes implicitly with the encode and decode functions for some integer N. If an element in M has the same component in each of the s-slots, then we call it a "diagonal" element. We let $\mathsf{Diag}(x)$ for $x \in \mathbb{F}_{p^k}$ denote the element $(x, x, \ldots, x) \in (\mathbb{F}_{p^k})^s$.

Our cryptosystem consists of a tuple $(\mathsf{ParamGen}, \mathsf{KeyGen}, \mathsf{KeyGen}^*, \mathsf{Enc}, \mathsf{Dec})$ of algorithms defined below, and parametrized by a security parameter κ.

$\mathsf{ParamGen}(1^\kappa, M)$: This parameter generation algorithm outputs an integer N (as above), definitions of the encode and decode functions, and a description of a randomized algorithm D_ρ^d, which outputs vectors in $\mathbb{Z}^d$. We assume that D_ρ^d outputs $\mathbf{r}$ with $\|\mathbf{r}\|_\infty \leq \rho$, except with negligible probability. The algorithm D_ρ^d is used by the encryption algorithm to select the random coins needed during encryption. The algorithm $\mathsf{ParamGen}$ also outputs an additive abelian group G. The group G also possesses a (not necessarily closed) multiplicative operator, which is commutative and distributes over the additive group of G. The group G is the group in which the ciphertexts will be assumed to lie. We write $\boxplus$ and $\boxtimes$ for the operations on G, and extend these in the natural way to vectors and matrices of elements of G. Finally $\mathsf{ParamGen}$ outputs a set C of allowable arithmetic SIMD circuits over $(\mathbb{F}_{p^k})^s$, these are the set of functions which our scheme will be able to evaluate ciphertexts over. We can think of C as a subset of $\mathbb{F}_{p^k}[X_1, X_2, \ldots, X_n]$, where we evaluate a function $f \in \mathbb{F}_{p^k}[X_1, X_2, \ldots, X_n]$ a total of s times in parallel on inputs from $(\mathbb{F}_{p^k})^n$. We assume that all other algorithms take as implicit input the output $P \leftarrow (1^\kappa, N, \mathsf{encode}, \mathsf{decode}, \mathsf{D}_\rho^d, G, C)$ of $\mathsf{ParamGen}$.

$\mathsf{KeyGen}()$: This algorithm outputs a public key pk and a secret key sk.

$\mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}, \mathbf{r})$: On input of $\mathbf{x} \in \mathbb{Z}^N$, $\mathbf{r} \in \mathbb{Z}^d$, this deterministic algorithm outputs a ciphertext $c \in G$. When applying this algorithm one would obtain $\mathbf{x}$ from the application of the encode function, and $\mathbf{r}$ by calling D_ρ^d. This is what we mean when we write $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})$, where $\mathbf{m} \in M$. However, it is convenient for us to define Enc on the intermediate state, $\mathbf{x} = \mathsf{encode}(\mathbf{m})$. To ease notation we write $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{x})$ if the value of the randomness $\mathbf{r}$ is not important for our discussion. To make our zero-knowledge proofs below work, we will require that addition of V "clean" ciphertexts (for "small" values of V), of plaintext $\mathbf{x}_i$ in $\mathbb{Z}^N$, using randomness $\mathbf{r}_i$, results in a ciphertext which could be obtained by adding the plaintexts and randomness, as integer vectors, and then applying $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}, \mathbf{r})$, i.e.

$$\mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_1 + \cdots + \mathbf{x}_V, \mathbf{r}_1 + \cdots + \mathbf{r}_V) = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_1, \mathbf{r}_1) \boxplus \cdots \boxplus \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_V, \mathbf{r}_V).$$

$\mathsf{Dec}_{\mathsf{sk}}(c)$: On input the secret key and a ciphertext c it returns either an element $\mathbf{m} \in M$, or the symbol $\perp$.

We are now able to define various properties of the above abstract scheme that we will require. But first a bit of notation: For a function $f \in C$ we let $n(f)$ denote the number of variables in f, and we

let $\widehat{f}$ denote the function on G induced by f. That is, given f, we replace every + operation with a $\boxplus$, every $\cdot$ operation is replaced with a $\boxtimes$ and every constant c is replaced by $\mathsf{Enc}_{\mathsf{pk}}(\mathsf{encode}(c), \mathbf{0})$. Also, given a set of $n(f)$ vectors $\mathbf{x}_1, \ldots, \mathbf{x}_{n(f)}$, we define $f(\mathbf{x}_1, \ldots, \mathbf{x}_{n(f)})$ in the natural way by applying f in parallel on each coordinate.

Correctness: Intuitively correctness means that if one decrypts the result of a function $f \in C$ applied to $n(f)$ encrypted vectors of variables, then this should return the same value as applying the function to the $n(f)$ plaintexts. However, to apply the scheme in our protocol, we need to be a bit more liberal, namely the decryption result should be correct, even if the ciphertexts we start from were not necessarily generated by the normal encryption algorithm. They only need to "contain" encodings and randomness that are not too large, such that the encodings decode to legal values. Formally, the scheme is said to be (B_{plain}, B_{rand}, C)-correct if

$$\begin{aligned}\Pr[\, & P \leftarrow \mathsf{ParamGen}(1^\kappa, M), \quad (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(), \quad \text{for any } f \in C, \\ & \text{any } \mathbf{x}_i, \mathbf{r}_i, \text{ with } \|\mathbf{x}_i\|_\infty \le B_{plain}, \ \|\mathbf{r}_i\|_\infty \le B_{rand}, \ \mathsf{decode}(\mathbf{x}_i) \in (\mathbb{F}_{p^k})^s, \\ & i = 1, \ldots, n(f), \text{ and } c_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_i, \mathbf{r}_i), \ c \leftarrow \widehat{f}(c_1, \ldots, c_{n(f)}) : \\ & \mathsf{Dec}_{\mathsf{sk}}(c) \neq f(\mathsf{decode}(\mathbf{x}_1), \ldots, \mathsf{decode}(\mathbf{x}_{n(f)}))\,] \qquad < \epsilon(\kappa).\end{aligned}$$

We will say that a ciphertext is (B_{plain}, B_{rand}, C)-*admissible* if it can be obtained as the ciphertext c in the above experiment, i.e., by applying a function from C to ciphertexts generated from (legal) encodings and randomness that are bounded by B_{plain} and B_{rand}.

$\mathsf{KeyGen}^*()$: This is a randomized algorithm that outputs a *meaningless public key* $\widetilde{\mathsf{pk}}$. We require that an encryption of any message $\mathsf{Enc}_{\widetilde{\mathsf{pk}}}(\mathbf{x})$ is statistically indistinguishable from an encryption of 0. Furthermore, if we set $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$ and $\widetilde{\mathsf{pk}} \leftarrow \mathsf{KeyGen}^*()$, then pk and $\widetilde{\mathsf{pk}}$ are computationally indistinguishable. This implies the scheme is IND-CPA secure in the usual sense.

Distributed Decryption: We assume, as a set up assumption, that a common public key has been set up where the secret key has been secret-shared among the players in such a way that they can collaborate to decrypt a ciphertext. We assume throughout that only (B_{plain}, B_{rand}, C)-admissible ciphertexts are to be decrypted, this constraint is guaranteed by our main protocol.

We note that some set-up assumption is always required to show UC security which is our goal here. Concretely, we assume that a functionality $\mathcal{F}_{\text{KeyGen}}$ is available, as specified in Figure 2. It basically generates a key pair and secret-shares the secret key among the players using a secret-sharing scheme that is assumed to be given as part of the specification of the cryptosystem. Since we want to allow corruption of all but one player, the maximal unqualified sets must be all sets of $n-1$ players.

Functionality $\mathcal{F}_{\text{KeyGen}}$

1. When receiving "start" from all honest players, run $P \leftarrow \mathsf{ParamGen}(1^\kappa, M)$, and then, using the parameters generated, run $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$ (recall P, and hence 1^κ, is an implicit input to all functions we specify). Send pk to the adversary.
2. We assume a secret sharing scheme is given with which sk can be secret-shared. Receive from the adversary a set of shares s_j for each corrupted player P_j.
3. Construct a complete set of shares $(s_1, \ldots, s_n)$ consistent with the adversary's choices and sk. Note that this is always possible since the corrupted players form an unqualified set. Send pk to all players and s_i to each honest P_i.

Fig. 2. The Ideal Functionality for Distributed Key Generation

We note that it is possible to make a weaker set-up assumption, such as a common reference string (CRS), and using a general UC secure multiparty computation protocol for the CRS model to implement $\mathcal{F}_{\text{KeyGen}}$. While this may not be very efficient, one only needs to run this protocol once in the life-time of the system.

We also want our cryptosystem to implement the functionality $\mathcal{F}_{\text{KeyGenDec}}$ in Figure 3, which essentially specifies that players can cooperate to decrypt a (B_{plain}, B_{rand}, C)-admissible ciphertext, but the protocol is only secure against a passive attack: the adversary gets the correct decryption result, but can decide which result the honest players should learn.

Functionality $\mathcal{F}_{\text{KeyGenDec}}$

1. When receiving "start" from all honest players, run $\mathsf{ParamGen}(1^\kappa, M)$, and then, using the parameters generated, run $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$. Send pk to the adversary and to all players, and store sk.
2. Hereafter on receiving "decrypt c" for (B_{plain}, B_{rand}, C)-admissible c from all honest players, send c and $m \leftarrow \mathsf{Dec}_{\mathsf{sk}}(c)$ to the adversary. On receiving m' from the adversary, send "Result m'" to all players, Both m and m' may be a special symbol $\perp$ indicating that decryption failed.
3. On receiving "decrypt c to P_j" for admissible c, if P_j is corrupt, send $c, m \leftarrow \mathsf{Dec}_{\mathsf{sk}}(c)$ to the adversary. If P_j is honest, send c to the adversary. On receiving δ from the adversary, if $\delta \notin M$, send $\perp$ to P_j, if $\delta \in M$, send $\mathsf{Dec}_{\mathsf{sk}}(c) + \delta$ to P_j.

Fig. 3. The Ideal Functionality for Distributed Key Generation and Decryption

We are now finally ready to define the basic set of properties that the underlying cryptosystem should satisfy, in order to be used in our protocol. Here we use an "information theoretic" security parameter sec that controls the errors in our ZK proofs below.

Definition 1. (Admissible Cryptosystem.) *Let C contain formulas of form $(x_1 + \cdots + x_n) \cdot (y_1 + \cdots + y_n) + z_1 + \cdots + z_n$, as well as all "smaller" formulas , i.e., with a smaller number of additions and possibly no multiplication. A cryptosystem is* admissible *if it is defined by algorithms $(\mathsf{ParamGen}, \mathsf{KeyGen}, \mathsf{KeyGen}^*, \mathsf{Enc}, \mathsf{Dec})$ with properties as defined above, is (B_{plain}, B_{rand}, C)-correct, where*

$$B_{plain} = N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{(1/2+\nu)\mathsf{sec}}, \quad B_{rand} = d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{(1/2+\nu)\mathsf{sec}};$$

and where $\nu > 0$ can be an arbitrary constant. Finally there exist a secret sharing scheme as required in $\mathcal{F}_{\text{KeyGen}}$ and a protocol $\Pi_{\mathsf{KeyGenDec}}$ with the property that when composed with $\mathcal{F}_{\text{KeyGen}}$ it securely implements the functionality $\mathcal{F}_{\text{KeyGenDec}}$.

The set C is defined to contain all computations on ciphertext that we need in our main protocol. Throughout the paper we will assume that B_{plain}, B_{rand} are defined as here in terms of τ, ρ and sec. This is because these are the bounds we can force corrupt players to respect via our zero-knowledge protocol, as we shall see.

4 Zero-Knowledge Proof of Plaintext Knowledge

This section presents a zero-knowledge protocol that takes as input sec ciphertexts $c_1, \ldots, c_{\mathsf{sec}}$ generated by one of the players in our protocol, who will act as the prover. If the prover is honest then $c_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_i, \mathbf{r}_i)$, where $\mathbf{x}_i$ has been obtained from the encode function, i.e. $\|\mathbf{x}_i\|_\infty \leq \tau$, and $\mathbf{r}_i$

has been generated from D_ρ^d (so we may assume that $\|\mathbf{r}_i\|_\infty \le \rho$). Our protocol is a zero-knowledge proof of plaintext knowledge (ZKPoPK) for the following relation:

$$\begin{aligned} R_{\mathsf{PoPK}} = \{\ (x,w) |\ & x = (\mathsf{pk}, \mathbf{c}),\ w = ((\mathbf{x}_1, \mathbf{r}_1), \ldots, (\mathbf{x}_{\mathsf{sec}}, \mathbf{r}_{\mathsf{sec}})) : \\ & \mathbf{c} = (c_1, \ldots, c_{\mathsf{sec}}),\ c_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_i, \mathbf{r}_i), \\ & \|\mathbf{x}_i\|_\infty \le B_{plain},\ \mathsf{decode}(\mathbf{x}_i) \in (\mathbb{F}_{p^k})^s,\ \|\mathbf{r}_i\|_\infty \le B_{rand}\ \}\,. \end{aligned}$$

The zero-knowledge and completeness properties hold only if the ciphertexts c_i satisfy $\|\mathbf{x}_i\|_\infty \le \tau$ and $\|\mathbf{r}_i\|_\infty \le \rho$.

In our preprocessing protocol, players will be required to give such a ZKPoPK for all ciphertexts they provide. By admissibility of the cryptosystem, this will imply that every ciphertext occurring in the protocol will be (B_{plain}, B_{rand}, C)-admissible and can therefore be decrypted correctly. The ZKPoPK can also be called with a flag diag which will modify the proof so that it additionally proves that $\mathsf{decode}(\mathbf{x}_i)$ is a diagonal element.

The protocol is not meant to implement an ideal functionality, but we can still use it and prove UC security for the main protocol, since we will always generate the challenge $\mathbf{e}$ by calling the $\mathcal{F}_{\text{RAND}}$ ideal functionality (see Appendix E). Hence the honest-verifier ZK property implies straight-line simulation[5]. As for knowledge extraction, the UC simulator we construct in our security proof will know the secret key for the cryptosystem and can therefore extract a dishonest prover's witness simply by decrypting. In the reduction to show that the simulator works, we do not know the secret key, but here we are allowed to do extraction by rewinding.

The protocol and its proof of security are given in Appendix A.1, Figure 9 and its computational complexity per ciphertext is essentially the cost of a constant number of encryptions. In Appendix A.1, we also give a variant of the ZK proof that allows even smaller values for B_{plain}, B_{rand}, namely $B_{plain} = N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8}$, $B_{rand} = d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8}$, and hence improves performance further. This variant is most efficient when executed using the Fiat-Shamir heuristic (although it can also work without random oracles), and we believe this variant is the best for a practical implementation.

5 The Preprocessing Phase

In this section we construct the protocol Π_{PREP} which securely implements the functionality $\mathcal{F}_{\text{PREP}}$ (specified in Figure 16) in the presence of functionalities $\mathcal{F}_{\text{KEYGENDEC}}$ (Figure 3) and $\mathcal{F}_{\text{RAND}}$ (Figure 14). The preprocessing uses the above abstract cryptosystem with $M = (\mathbb{F}_{p^k})^s$, but the online phase is designed for messages in $\mathbb{F}_{p^k}$. Therefore, we extend the notation $\langle\cdot\rangle$ and $[\![\cdot]\!]$ to messages in M: since addition and multiplication on M are componentwise, for $\mathbf{m} = (m_1, \ldots, m_s)$, we define $\langle\mathbf{m}\rangle = (\langle m_1\rangle, \ldots, \langle m_s\rangle)$ and similarly for $[\![\mathbf{m}]\!]$. Conversely, once a representation (or a pair, triple) on vectors is produced in the preprocessing, it will be disassembled into its coordinates, so that it can be used in the online phase. In Figures 4,5 and 6, we introduce subprotocols that are accessed by the main preprocessing protocol in several steps. Note that the subprotocols are not meant to implement ideal functionalities: their purpose is merely to summarize parts of the main protocol that are repeated in various occasions. Theorem 3 below is proved in Appendix A.5.

[5] $\mathcal{F}_{\text{RAND}}$ can be implemented by standard methods, and the complexity of this is not significant for the main protocol since we may use the same challenge for many instances of the proof, and each proof handles sec ciphertexts.

Theorem 3. *The protocol* Π_{PREP} *(Figure 7) implements* $\mathcal{F}_{\text{PREP}}$ *with computational security against any static, active adversary corrupting up to* $n-1$ *parties, in the* $\mathcal{F}_{\text{KeyGen}}, \mathcal{F}_{\text{Rand}}$*-hybrid model when the underlying cryptosystem is admissible*[6].

Protocol Reshare

Usage: Input is $e_{\mathbf{m}}$, where $e_{\mathbf{m}} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})$ is a public ciphertext and a parameter *enc*, where *enc* = NewCiphertext or *enc* = NoNewCiphertext. Output is a share $\mathbf{m}_i$ of $\mathbf{m}$ to each player P_i; and if *enc* = NewCiphertext, a ciphertext $e'_{\mathbf{m}}$. The idea is that $e_{\mathbf{m}}$ could be a product of two ciphertexts, which Reshare converts to a "fresh" ciphertext $e'_{\mathbf{m}}$. Since Reshare uses distributed decryption (that may return an incorrect result), it is not guaranteed that $e_{\mathbf{m}}$ and $e'_{\mathbf{m}}$ contain the same value, but it *is* guaranteed that $\sum_i \mathbf{m}_i$ is the value contained in $e'_{\mathbf{m}}$.

Reshare$(e_{\mathbf{m}}, enc)$:

1. Each player P_i samples a uniform $\mathbf{f}_i \in (\mathbb{F}_{p^k})^s$. Define $\mathbf{f} := \sum_{i=1}^n \mathbf{f}_i$.
2. Each player P_i computes and broadcasts $e_{\mathbf{f}_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{f}_i)$.
3. Each player P_i runs Π_{ZKPoPK} acting as a prover on $e_{\mathbf{f}_i}$. The protocol aborts if any proof fails.
4. The players compute $e_{\mathbf{f}} \leftarrow e_{\mathbf{f}_1} \boxplus \cdots \boxplus e_{\mathbf{f}_n}$, and $e_{\mathbf{m}+\mathbf{f}} \leftarrow e_{\mathbf{m}} \boxplus e_{\mathbf{f}}$.
5. The players invoke $\mathcal{F}_{\text{KeyGenDec}}$ to decrypt $e_{\mathbf{m}+\mathbf{f}}$ and thereby obtain $\mathbf{m}+\mathbf{f}$.
6. P_1 sets $\mathbf{m}_1 \leftarrow \mathbf{m}+\mathbf{f}-\mathbf{f}_1$, and each player P_i $(i \neq 1)$ sets $\mathbf{m}_i \leftarrow -\mathbf{f}_i$.
7. If *enc* = NewCiphertext, all players set $e'_{\mathbf{m}} \leftarrow \mathsf{Enc}_{pk}(\mathbf{m}+\mathbf{f}) \boxminus e_{\mathbf{f}_1} \boxminus \cdots \boxminus e_{\mathbf{f}_n}$, where a default value for the randomness is used when computing $\mathsf{Enc}_{pk}(\mathbf{m}+\mathbf{f})$.

Fig. 4. The sub-protocol for additively secret sharing a plaintext $\mathbf{m} \in (\mathbb{F}_{p^k})^s$ on input a ciphertext $e_{\mathbf{m}} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})$.

Protocol PBracket

Usage: On input shares $\mathbf{v}_1, \ldots, \mathbf{v}_n$ privately held by the players and public ciphertext $e_{\mathbf{v}}$, this protocol generates $[\![\mathbf{v}]\!]$. It is assumed that $\sum_i \mathbf{v}_i$ is the plaintext contained in $e_{\mathbf{v}}$.

PBracket$(\mathbf{v}_1, \ldots, \mathbf{v}_n, e_{\mathbf{v}})$:

1. For $i = 1, \ldots, n$
   (a) All players set $e_{\gamma_i} \leftarrow e_{\beta_i} \boxtimes e_{\mathbf{v}}$ (note that e_{β_i} is generated during the initialization process, and known by every player)
   (b) Players generate $(\gamma_i^1, \ldots \gamma_i^n) \leftarrow$ Reshare$(e_{\gamma_i}, \text{NoNewCiphertext})$, so each player P_j gets a share γ_i^j of $\mathbf{v} \cdot \beta_i$.
2. Output the representation $[\![\mathbf{v}]\!] = (\mathbf{v}_1, \ldots, \mathbf{v}_n, (\beta_i, \gamma_1^i, \ldots, \gamma_n^i)_{i=1,\ldots,n})$.

Fig. 5. The sub-protocol for generating $[\![\mathbf{v}]\!]$.

Protocol PAngle

Usage: On input shares $\mathbf{v}_1, \ldots, \mathbf{v}_n$ privately held by the players and public ciphertext $e_{\mathbf{v}}$, this protocol generates $\langle \mathbf{v} \rangle$. It is assumed that $\sum_i \mathbf{v}_i$ is the plaintext contained in $e_{\mathbf{v}}$.

PAngle$(\mathbf{v}_1, \ldots, \mathbf{v}_n, e_{\mathbf{v}})$:

1. All players set $e_{\mathbf{v}\cdot\alpha} \leftarrow e_{\mathbf{v}} \boxtimes e_{\alpha}$ (note that e_{α} is generated during the initialization process, and known by every player)
2. Players generate $(\gamma_1, \ldots, \gamma_n) \leftarrow$ Reshare$(e_{\mathbf{v}\cdot\alpha}, \text{NoNewCiphertext})$, so each player P_i gets a share γ_i of $\alpha \cdot \mathbf{v}$.
3. Output representation $\langle \mathbf{v} \rangle = (0, \mathbf{v}_1, \ldots, \mathbf{v}_n, \gamma_1, \ldots, \gamma_n)$.

Fig. 6. The sub-protocol for generating $\langle \mathbf{v} \rangle$.

[6] The definition of admissible cryptosystem demands a decryption protocol that implements $\mathcal{F}_{\text{KeyGenDec}}$ based on $\mathcal{F}_{\text{KeyGen}}$, hence the theorem only assumes $\mathcal{F}_{\text{KeyGen}}$.

Protocol Π_{PREP}

Usage: The Triple-step is always executed sec times in parallel. This ensures that when calling Π_{ZKPoPK}, we can always give it the sec ciphertexts it requires as input. In addition both Π_{ZKPoPK} and Π_{PREP} can be executed in a SIMD fashion, i.e. they are data-oblivious bar when they detect an error. Thus we can execute Π_{ZKPoPK} and Π_{PREP} on the packed plaintext space $(\mathbb{F}_{p^k})^s$. Thereby, we generate $s \cdot \mathsf{sec}$ elements in one go and then buffer the generated triples, outputting the next unused one on demand.

Initialize: This step generates the global key α and "personal keys" β_i.

1. The players call "start" on $\mathcal{F}_{\text{KeyGenDec}}$ to obtain the public key pk
2. Each player P_i generates a MAC-key $\beta_i \in \mathbb{F}_{p^k}$
3. Each player P_i generates $\alpha_i \in \mathbb{F}_{p^k}$. Let $\alpha := \sum_{i=1}^{n} \alpha_i$
4. Each player P_i computes and broadcasts $e_{\alpha_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathsf{Diag}(\alpha_i))$, $e_{\beta_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathsf{Diag}(\beta_i))$
5. Each player P_i invokes Π_{ZKPoPK} (with diag set to true) acting as prover on input $(e_{\alpha_i}, \ldots, e_{\alpha_i})$ and on input $(e_{\beta_i}, \ldots, e_{\beta_i})$, where $e_{\alpha_i}, e_{\beta_i}$ are repeated sec times, which is the number of ciphertexts Π_{ZKPoPK} requires as input. (This is not very efficient, but only needs to be done once for each player.)
6. All players compute $e_\alpha \leftarrow e_{\alpha_1} \boxplus \cdots \boxplus e_{\alpha_n}$, and generate $[\![\mathsf{Diag}(\alpha)]\!] \leftarrow \mathsf{PBracket}(\mathsf{Diag}(\alpha_1), \ldots, \mathsf{Diag}(\alpha_n), e_\alpha)$

Pair: This step generates a pair $[\![\mathbf{r}]\!], \langle \mathbf{r} \rangle$, and can be used to generate a single value $[\![\mathbf{r}]\!]$, by not performing the call to Pangle

1. Each player P_i generates $\mathbf{r}_i \in (\mathbb{F}_{p^k})^s$. Let $\mathbf{r} := \sum_{i=1}^{n} \mathbf{r}_i$
2. Each player P_i computes and broadcasts $e_{\mathbf{r}_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{r}_i)$. Let $e_{\mathbf{r}} = e_{\mathbf{r}_1} \boxplus \cdots \boxplus e_{\mathbf{r}_n}$
3. Each player P_i invokes Π_{ZKPoPK} acting as prover on the ciphertext he generated
4. Players generate $[\![\mathbf{r}]\!] \leftarrow \mathsf{PBracket}(\mathbf{r}_1, \ldots, \mathbf{r}_n, e_{\mathbf{r}})$, $\langle \mathbf{r} \rangle \leftarrow \mathsf{PAngle}(\mathbf{r}_1, \ldots, \mathbf{r}_n, e_{\mathbf{r}})$

Triple: This step generates a multiplicative triple $\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, \langle \mathbf{c} \rangle$

1. Each player P_i generates $\mathbf{a}_i, \mathbf{b}_i \in (\mathbb{F}_{p^k})^s$. Let $\mathbf{a} := \sum_{i=1}^{n} \mathbf{a}_i$, $\mathbf{b} := \sum_{i=1}^{n} \mathbf{b}_i$
2. Each player P_i computes and broadcasts $e_{\mathbf{a}_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{a}_i)$, $e_{\mathbf{b}_i} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{b}_i)$
3. Each player P_i invokes Π_{ZKPoPK} acting as prover on the ciphertexts he generated.
4. The players set $e_{\mathbf{a}} \leftarrow e_{\mathbf{a}_1} \boxplus \cdots \boxplus e_{\mathbf{a}_n}$ and $e_{\mathbf{b}} \leftarrow e_{\mathbf{b}_1} \boxplus \cdots \boxplus e_{\mathbf{b}_n}$
5. Players generate $\langle \mathbf{a} \rangle \leftarrow \mathsf{PAngle}(\mathbf{a}_1, \ldots, \mathbf{a}_n, e_{\mathbf{a}})$, $\langle \mathbf{b} \rangle \leftarrow \mathsf{PAngle}(\mathbf{b}_1, \ldots, \mathbf{b}_n, e_{\mathbf{b}})$.
6. All players compute $e_{\mathbf{c}} \leftarrow e_{\mathbf{a}} \boxtimes e_{\mathbf{b}}$
7. Players set $(\mathbf{c}_1, \ldots, \mathbf{c}_n, e'_{\mathbf{c}}) \leftarrow \mathsf{Reshare}(e_{\mathbf{c}}, \mathsf{NewCiphertext})$.
8. Players generate $\langle \mathbf{c} \rangle \leftarrow \mathsf{PAngle}(\mathbf{c}_1, \ldots, \mathbf{c}_n, e'_{\mathbf{c}})$.

Fig. 7. The protocol for constructing the global key $[\![\alpha]\!]$, pairs $[\![\mathbf{r}]\!], \langle \mathbf{r} \rangle$ and multiplicative triples $\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, \langle \mathbf{c} \rangle$.

6 Concrete Instantiation of the Abstract Scheme based on LWE

We now describe the concrete scheme, which is based on the somewhat homomorphic encryption scheme of Brakerski and Vaikuntanathan (BV) [7]. The main differences are that we are only interested in evaluation of circuits of multiplicative depth one, we are interested in performing operations in parallel on multiple data items, and we require a distributed decryption procedure. In this section we detail the scheme and the distributed decryption procedure; in Appendix D we discuss security of the scheme, and present some sample parameter sizes and performance figures.

$\underline{\mathsf{ParamGen}(1^\kappa, M)}$: Recall the message space is given by $M = (\mathbb{F}_{p^k})^s$ for two integers k and s, and a prime p, i.e. the message space is s copies of the finite field $\mathbb{F}_{p^k}$. To map this to our scheme below, one first finds a cyclotomic polynomial $F(X) := \Phi_m(X)$ of degree $N := \phi(m)$, where N is lower bounded by some function of the security parameter κ. The polynomial $F(X)$ needs to be such that modulo p the polynomial $F(X)$ factors into l' irreducible factors of degree k' where $l' \geq s$ and k divides k'. We then define an algebra A_p as $A_p := \mathbb{F}_p[X]/F(X)$ and we have an embedding of M into A_p, $\phi : M \to A_p$. By "lifting" modulo p we see that there is a natural inclusion $\iota : A_p \to \mathbb{Z}^N$, which maps the polynomial of degree less than N with coefficients in $\mathbb{F}_p$ into the integer vector of length N with coefficients in the range $(-p/2, \ldots, p/2]$. The encode function is then defined by

$\iota(\phi(\mathbf{m}))$ for $\mathbf{m} \in (\mathbb{F}_{p^k})^s$, with decode defined by $\phi^{-1}(\mathbf{x} \pmod p)$ for $\mathbf{x} \in \mathbb{Z}^N$. It is clear, by choice of the natural inclusion ι, that $\|\mathsf{encode}(\mathbf{m})\|_\infty \leq p/2 = \tau$.

We pick a large integer q, whose size we will determine later, and defined $A_q := (\mathbb{Z}/q\mathbb{Z})[X]/F(X)$, i.e. the ring of integer polynomials modulo reduction by $F(X)$ and q. In practice we consider the image of encode to lie in A_q, and thus we abuse notation, by writing addition and multiplication in A_q by $+$ and $\cdot$. Note, that this means that applying decode to elements obtained from encode followed by a series of arithmetic operations may not result in the value in M which one would expect. This corresponds to where our scheme can only evaluate circuits from a given set C.

The ciphertext space G is defined to be A_q^3, with addition $\boxplus$ defined componentwise. The multiplicative operator $\boxtimes$ is defined as follows

$$(\mathbf{a}_0, \mathbf{a}_1, 0) \boxtimes (\mathbf{b}_0, \mathbf{b}_1, 0) := (\mathbf{a}_0 \cdot \mathbf{b}_0, \mathbf{a}_1 \cdot \mathbf{b}_0 + \mathbf{a}_0 \cdot \mathbf{b}_1, -\mathbf{a}_1 \cdot \mathbf{b}_1),$$

i.e. multiplication is only defined on elements whose third coefficient is zero.

We define D_ρ^d as follows: The discrete Gaussian $D_{\mathbb{Z}^N,s}$, with *Gaussian parameter* s, is defined to be the random variable on $\mathbb{Z}_q^N$ (centered around the origin) obtained from sampling $\mathbf{x} \in \mathbb{R}^N$, with probability proportional to $\exp(-\pi \cdot \|\mathbf{x}\|_2/s^2)$, and then rounding the result to the nearest lattice point and reducing it modulo q. Note, sampling from the distribution with probability density function proportional to $\exp(-\pi \cdot \|\mathbf{x}\|_2/s^2)$, means using a normal variate with mean zero, and standard deviation $r := s/\sqrt{2 \cdot \pi}$. In our concrete scheme we set $d := 3 \cdot N$ and define D_ρ^d to be the distribution defined by $(D_{\mathbb{Z}^N,s})^3$. Note, that in the notation D_ρ^d the implicit dependence on q has been suppressed to ease readability. The determining of q and r as functions of all the other parameters, we leave until we discuss security of the scheme.

$\underline{\mathsf{KeyGen}()}$: We will use the public key version of the Brakerski–Vaikuntanathan scheme [7]. Given the above set up, key generation proceeds as follows: First one samples elements $\mathbf{a} \leftarrow A_q$ and $\mathbf{s}, \mathbf{e} \leftarrow D_{\mathbb{Z}^N,s}$. Then treating $\mathbf{s}$ and $\mathbf{e}$ as elements of A_q one computes $\mathbf{b} \leftarrow (\mathbf{a} \cdot \mathbf{s}) + (p \cdot \mathbf{e})$. The public and private key are then set to be $\mathsf{pk} \leftarrow (\mathbf{a}, \mathbf{b})$ and $\mathsf{sk} \leftarrow \mathbf{s}$.
$\underline{\mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}, \mathbf{r})}$: Given a message $\mathbf{x} \leftarrow \mathsf{encode}(m)$ where $m \in M$, and $\mathbf{r} \in \mathsf{D}_\rho^d$, we proceed as follows: The element $\mathbf{r}$ is parsed as $(\mathbf{u}, \mathbf{v}, \mathbf{w}) \in (\mathbb{Z}^N)^3$. Then the encryptor computes $\mathbf{c}_0 \leftarrow (\mathbf{b} \cdot \mathbf{v}) + (p \cdot \mathbf{w}) + \mathbf{x}$ and $\mathbf{c}_1 \leftarrow (\mathbf{a} \cdot \mathbf{v}) + (p \cdot \mathbf{u})$. Finally returning the ciphertext $(\mathbf{c}_0, \mathbf{c}_1, 0)$.
$\underline{\mathsf{Dec}_{\mathsf{sk}}(c)}$: Given a secret key $\mathsf{sk} = \mathbf{s}$ and a ciphertext $c = (\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2)$ this algorithm computes the element in A_q satisfying $\mathbf{t} = \mathbf{c}_0 - (\mathbf{s} \cdot \mathbf{c}_1) - (\mathbf{s} \cdot \mathbf{s} \cdot \mathbf{c}_2)$. On reduction by q the value of $\|\mathbf{t}\|_\infty$ will be bounded by a relatively small constant B; assuming of course that the "noise" within a ciphertext has not grown too large. We shall refer to the value $\mathbf{t} \mod q$ as the "noise", despite it also containing the message to be decrypted. At this point the decryptor simply reduces $\mathbf{t}$ modulo p to obtain the desired plaintext in A_q, which can then be decoded via the decode algorithm.
$\underline{\mathsf{KeyGen}^*()}$: This simply samples $\widehat{\mathbf{a}}, \widehat{\mathbf{b}} \leftarrow A_q$ and returns $\widehat{\mathsf{pk}} := (\widehat{\mathbf{a}}, \widehat{\mathbf{b}})$.

Following the discussion in [7] we see that with this *fixed* ciphertext space, our scheme is somewhat homomorphic. It can support a relatively large number of addition operations, and a single multiplication.

Distributed Version We now extend the scheme above to enable distributed decryption. We first set up the distributed keys as follows. After invoking the functionality for key generation, each player obtains a share $\mathsf{sk}_i = (\mathbf{s}_{i,1}, \mathbf{s}_{i,2})$, these are chosen uniformly such that the master secret is written

as

$$\mathbf{s} = \mathbf{s}_{1,1} + \cdots + \mathbf{s}_{n,1}, \quad \mathbf{s} \cdot \mathbf{s} = \mathbf{s}_{1,2} + \cdots + \mathbf{s}_{n,2}.$$

As remarked earlier this one-time setup procedure can be accomplished by standard UC-secure multiparty computation protocols such as that described in [5]. The following theorem is proved in Appendix A.6. It depends on the constant B defined above. In Appendix D we compute the value of B when the input ciphertext is (B_{plain}, B_{rand}, C)-admissible, and show how to choose parameters for the cryptosystem such that the required bound on B is satisfied.

Theorem 4. *In the $\mathcal{F}_{\text{KeyGen}}$-hybrid model, the protocol Π_{DDec} (Figure 8) implements $\mathcal{F}_{\text{KeyGenDec}}$ with statistical security against any static active adversary corrupting up to $n-1$ parties if $B + 2^{\mathsf{sec}} \cdot B < q/2$.*

Protocol Π_{DDec}

Initialize: Each party P_i on being given the ciphertext $c = (\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2)$, and an upper bound B on the infinity norm of $\mathbf{t}$ above, computes

$$\mathbf{v}_i \leftarrow \begin{cases} \mathbf{c}_0 - (\mathbf{s}_{i,1} \cdot \mathbf{c}_1) - (\mathbf{s}_{i,2} \cdot \mathbf{c}_2) & \text{if } i = 1 \\ -(\mathbf{s}_{i,1} \cdot \mathbf{c}_1) - (\mathbf{s}_{i,2} \cdot \mathbf{c}_2) & \text{if } i \neq 1 \end{cases}$$

and sets $\mathbf{t}_i \leftarrow \mathbf{v}_i + p \cdot \mathbf{r}_i$ where $\mathbf{r}_i$ is a random element with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$.

Public Decryption: All the players are supposed to learn the message.

- Each party P_i broadcasts $\mathbf{t}_i$
- All players compute $\mathbf{t}' \leftarrow \mathbf{t}_1 + \cdots + \mathbf{t}_n$ and obtain a message $m' \leftarrow \mathsf{decode}(\mathbf{t}' \bmod p)$.

Private Decryption: Only player P_j is supposed to learn the message.

- Each party P_i sends $\mathbf{t}_i$ to P_j
- P_j computes $\mathbf{t}' \leftarrow \mathbf{t}_1 + \cdots + \mathbf{t}_n$ and obtain a message $m' \leftarrow \mathsf{decode}(\mathbf{t}' \bmod p)$.

Fig. 8. The distributed decryption protocol.

7 Acknowledgements

The first, second and fourth author acknowledge support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, within which [part of] this work was performed; and also from the CFEM research center (supported by the Danish Strategic Research Council) within which part of this work was performed.

The third author was supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant COED–EP/I03126X, the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079, and by a Royal Society Wolfson Merit Award. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, AFRL, the U.S. Government, the European Commission or EPSRC.

The authors would like to thank Robin Chapman, Henri Cohen and Rob Harley for various discussions whilst this work was carried out.

References

1. S. Arora and R. Ge. New algorithms for learning in presence of errors. In L. Aceto, M. Henzinger, and J. Sgall, editors, *ICALP (1)*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011.
2. G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold fhe. In Pointcheval and Johansson [26], pages 483–501.
3. D. Beaver. Efficient multiparty protocols using circuit randomization. In J. Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.
4. E. Ben-Sasson, S. Fehr, and R. Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. *IACR Cryptology ePrint Archive*, 2011:629, 2011.
5. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, pages 169–188, 2011.
6. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:111, 2011.
7. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In P. Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.
8. R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, pages 494–503, 2002.
9. Y. Chen and P. Q. Nguyen. Bkz 2.0: Better lattice security estimates. In D. H. Lee and X. Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.
10. R. Cramer, I. Damgård, and V. Pastro. On the amortized complexity of zero knowledge protocols for multiplicative relations. In *ICITS*, 2012. To appear.
11. I. Damgård, M. Keller, E. Larraia, C. Miles, and N. P. Smart. Implementing aes via an actively/covertly secure dishonest-majority mpc protocol. *IACR Cryptology ePrint Archive*, 2012:262, 2012.
12. I. Damgård and J. B. Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In M. Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 581–596. Springer, 2002.
13. I. Damgård and C. Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In *CRYPTO*, pages 558–576, 2010.
14. N. Gama and P. Q. Nguyen. Predicting lattice reduction. In N. P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 2008.
15. C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.
16. C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In Pointcheval and Johansson [26], pages 465–482.
17. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge from secure multiparty computation. In D. S. Johnson and U. Feige, editors, *STOC*, pages 21–30. ACM, 2007.
18. Y. Ishai, M. Prabhakaran, and A. Sahai. Founding cryptography on oblivious transfer - efficiently. In D. Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591. Springer, 2008.
19. Y. Lindell. Highly-efficient universally-composable commitments based on the ddh assumption. In *EUROCRYPT*, pages 446–466, 2011.
20. R. Lindner and C. Peikert. Better key sizes (and attacks) for lwe-based encryption. In A. Kiayias, editor, *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, 2011.
21. V. Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In M. Matsui, editor, *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009.
22. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. 2011. Manuscript.
23. D. Micciancio and O. Regev. Lattice-based cryptography, 2008.
24. S. Myers, M. Sergi, and abhi shelat. Threshold fully homomorphic encryption and secure computation. *IACR Cryptology ePrint Archive*, 2011:454, 2011.
25. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. *IACR Cryptology ePrint Archive*, 2011:91, 2011.
26. D. Pointcheval and T. Johansson, editors. *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*. Springer, 2012.

27. M. Püschel and J. M. F. Moura. Algebraic signal processing theory: Cooley-tukey type algorithms for dcts and dsts. *IEEE Transactions on Signal Processing*, 56(4):1502–1521, 2008.
28. N. P. Smart and F. Vercauteren. Fully homomorphic simd operations. *IACR Cryptology ePrint Archive*, 2011:133, 2011.
29. S. Winkler and J. Wullschleger. On the efficiency of classical and quantum oblivious transfer reductions. In *CRYPTO*, pages 707–723, 2010.

A Proofs

A.1 Zero-Knowledge Proof

Construction of the Protocol. We will give two versions of the protocol. The first is a standard 3-move protocol, the second uses an "abort" technique to optimize the parameter values, this one is best suited for use with the Fiat-Shamir heuristic, and may be the best option for a practical implementation.

For the protocol, we will need that $\tau = p/2$, so that $\|\mathsf{encode}(\mathbf{m})\|_\infty \leq \tau = p/2$. This means that each entry in $\mathsf{encode}(\mathbf{m})$ corresponds to a uniquely determined residue mod p (or equivalently an element in $\mathbb{Z}_p$) and conversely each such residue is uniquely determined by $\mathbf{m}$. We did not ask for this in the abstract description, but the concrete instantiation satisfies this. Note that one problem we need to address in the protocol is that not all vectors in the input domain of decode will give us results in $\mathbb{F}_{p^k}$. However, if an input is equivalent mod p to $\mathsf{encode}(\mathbf{m})$ for some $\mathbf{m}$ then this is indeed the case, since then decode will return $\mathbf{m}$. Therefore the verifier explicitly checks whether the encodings the prover sends him decode to legal values, this will imply that the ciphertexts in question also decode to legal values.

We let R denote the matrix in $\mathbb{Z}^{\mathsf{sec}\times d}$ whose ith row is $\mathbf{r}_i$. It makes use of a matrix $M_{\mathbf{e}}$ defined as follows. Let $V := 2 \cdot \mathsf{sec} - 1$. For $\mathbf{e} \in \{0,1\}^{\mathsf{sec}}$ we define $M_{\mathbf{e}} \in \mathbb{Z}^{V\times \mathsf{sec}}$ to be the matrix whose (i,k)-th entry is given by $\mathbf{e}_{i-k+1}$, for $1 \leq i-k+1 \leq \mathsf{sec}$ and 0 otherwise.

Protocol Π_{ZKPoPK}

- For $i = 1,\ldots,V$, the prover sets $\mathbf{y}_i \leftarrow \mathbb{Z}^N$ and $\mathbf{s}_i \leftarrow \mathbb{Z}^d$, such that $\|\mathbf{y}_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$ and $\|\mathbf{s}_i\|_\infty \leq d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$. For $\mathbf{y}_i$, this is done as follows: choose a random message $\mathbf{m}_i \in (\mathbb{F}_{p^k})^s$ and set $\mathbf{y}_i = \mathsf{encode}(\mathbf{m}_i) + \mathbf{u}_i$, where each entry in $\mathbf{u}_i$ is a multiple of p, chosen uniformly at random, subject to $\|\mathbf{y}_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$. If diag is set to true, then the $\mathbf{m}_i$ are chosen to be diagonal elements.
- The prover computes $a_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i)$, for $i = 1,\ldots,V$, and defines $S \in \mathbb{Z}^{V\times d}$ to be the matrix whose ith row is $\mathbf{s}_i$ and sets $\mathbf{y} \leftarrow (\mathbf{y}_1,\ldots,\mathbf{y}_V)$, $\mathbf{a} \leftarrow (a_1,\ldots,a_V)$.
- The prover sends $\mathbf{a}$ to the verifier.
- The verifier selects $\mathbf{e} \in \{0,1\}^{\mathsf{sec}}$ and sends it to the prover.
- The prover sets $\mathbf{z} \leftarrow (\mathbf{z}_1,\ldots,\mathbf{z}_V)$, such that $\mathbf{z}^\mathsf{T} = \mathbf{y}^\mathsf{T} + M_{\mathbf{e}} \cdot \mathbf{x}^\mathsf{T}$, and $T = S + M_e \cdot R$. The prover sends $(\mathbf{z}, T)$ to the verifier.
- The verifier computes $d_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i)$, for $i = 1,\ldots,V$, where $\mathbf{t}_i$ is the ith row of T and sets $\mathbf{d} \leftarrow (d_1,\ldots,d_V)$.
- The verifier checks that $\mathsf{decode}(\mathbf{z}_i) \in \mathbb{F}_{p^k}^s$ and whether the following three conditions hold; he rejects if not

$$\mathbf{d}^\mathsf{T} = \mathbf{a}^\mathsf{T} \boxplus \left(M_{\mathbf{e}} \boxtimes \mathbf{c}^\mathsf{T}\right), \quad \|\mathbf{z}_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}, \quad \|\mathbf{t}_i\|_\infty \leq d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}.$$

- If diag is set to true the verifier also checks whether $\mathsf{decode}(\mathbf{z}_i)$ is a diagonal element, and rejects if it is not.

Fig. 9. The ZKPoPK Protocol, interactive version.

Theorem 5. *The protocol Π_{ZKPoPK} (Appendix A.1, Figure 9) is an honest-verifier zero-knowledge proof of knowledge for the relation R_{PoPK}.*

Proof (Theorem 5).

Completeness: Assume the prover is honest. For $i = 1, \ldots, V$ the verifier checks if $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i)$ equals $a_i \boxplus M_{\mathbf{e},i} \cdot \mathbf{c}^{\mathsf{T}}$, since $M_{\mathbf{e},i}$ is a scalar matrix we write multiplication with $\cdot$ as opposed to $\boxtimes$. The check passes because of the following relation:

$$\begin{aligned} a_i \boxplus \left(M_{\mathbf{e},i} \cdot \mathbf{c}^{\mathsf{T}}\right) &= \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i) \boxplus_{k=1}^{\mathsf{sec}} (M_{\mathbf{e},i,k} \cdot c_k) \\ &= \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i) \boxplus_{k=1}^{\mathsf{sec}} (M_{\mathbf{e},i,k} \cdot \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_k, \mathbf{r}_k)) \\ &= \mathsf{Enc}_{\mathsf{pk}}\left(\mathbf{y}_i + \sum_{k=1}^{\mathsf{sec}} M_{\mathbf{e},i,k} \cdot \mathbf{x}_k, \quad \mathbf{s}_i + \sum_{k=1}^{\mathsf{sec}} M_{\mathbf{e},i,k} \cdot \mathbf{r}_k\right) \\ &= \mathsf{Enc}_{\mathsf{pk}}\left(\mathbf{y}_i + M_{\mathbf{e},i} \cdot \mathbf{x}^{\mathsf{T}}, \quad \mathbf{s}_i + M_{\mathbf{e},i} \cdot \mathbf{r}^{\mathsf{T}}\right) = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i). \end{aligned}$$

Moreover, given that $\mathbf{z}_i = \mathbf{y}_i + M_{\mathbf{e},i} \cdot \mathbf{x}^{\mathsf{T}}$ and that all ciphertexts in $\mathbf{c}$ are (τ, ρ)-ciphertexts, we get that each single coordinate in $M_{\mathbf{e},i} \cdot \mathbf{x}^{\mathsf{T}}$ is numerically at most $\mathsf{sec} \cdot \tau$. Each coordinate of $\mathbf{y}_i$ was chosen from an interval that is a factor $N \cdot \mathsf{sec} \cdot 2^{\nu\mathsf{sec}-1}$ larger. By a union bound bound over the $N \cdot \mathsf{sec}$ coordinates involved, each coordinate in $\mathbf{z}_i$ fails to be in the required range with probability exponentially small in sec. A similar argument shows that the check $\|\mathbf{t}_i\|_\infty$ also fails with negligible probability. Finally, each $\mathbf{y}_i$ was constructed to be congruent mod p to the encoding of a value in $\mathbb{F}_{p^k}^s$. Since this is also the case for the $\mathbf{x}_i$'s if the prover is honest, the same is true for the $\mathbf{z}_i$'s, and they therefore decode to a value in $\mathbb{F}_{p^k}^s$. If diag was set to true, all $\mathbf{x}_i, \mathbf{y}_i$ contain diagonal plaintexts, and then the same is true for the $\mathbf{z}_i$.

Soundness: We consider a prover making a verifier accept both $(x, \mathbf{a}, \mathbf{e}, (\mathbf{z}, T))$ and $(x, \mathbf{a}, \mathbf{e}', (\mathbf{z}', T'))$ with $\mathbf{e} \neq \mathbf{e}'$. Since both checks $\mathbf{d}^{\mathsf{T}} = \mathbf{a}^{\mathsf{T}} \boxplus (M_{\mathbf{e}} \cdot \mathbf{c}^{\mathsf{T}})$ and $\mathbf{d}'^{\mathsf{T}} = \mathbf{a}^{\mathsf{T}} \boxplus (M_{\mathbf{e}'} \cdot \mathbf{c}^{\mathsf{T}})$ passed, one can subtract the two equalities and obtain

$$(M_{\mathbf{e}} - M_{\mathbf{e}'}) \boxtimes \mathbf{c}^{\mathsf{T}} = \left(\mathbf{d} \boxminus \mathbf{d}'\right)^{\mathsf{T}} \tag{1}$$

In order to find $\mathbf{x}$ and R such that $c_k = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_k, \mathbf{r}_k)$ for $k = 1, \ldots, \mathsf{sec}$, we first solve (1) as a linear system in $\mathbf{c}$. Let j be the highest index such that $\mathbf{e}_j \neq \mathbf{e}'_j$. The $\mathsf{sec} \times \mathsf{sec}$ submatrix of $M_{\mathbf{e}} - M_{\mathbf{e}'}$, consisting of the rows of $M_{\mathbf{e}} - M_{\mathbf{e}'}$ between j and $j + \mathsf{sec} - 1$ both included, is upper triangular with entries in $\{-1, 0, 1\}$ and its diagonal consists of the non-zero value $\mathbf{e}_j - \mathbf{e}'_j$ (so it is possible to find a solution for $\mathbf{c}$). Since the verifier has values $\mathbf{z}_i, \mathbf{t}_i, \mathbf{z}'_i, \mathbf{t}'_i$ such that $d_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i)$ and $d'_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}'_i, \mathbf{t}'_i)$, and given that $c_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_i, \mathbf{r}_i)$, it is possible to directly solve the linear system in $\mathbf{x}$ and R (since the cryptosystem is additively homomorphic), from the bottom equation to the one "in the middle" with index $\mathsf{sec}/2$. Since $\|\mathbf{z}_i\|_\infty, \|\mathbf{z}'_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$ and $\|\mathbf{t}_i\|_\infty, \|\mathbf{t}'_i\|_\infty \leq d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$, we conclude that $c_{\mathsf{sec}-i}$ is a $(s \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}+i}, d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}+i})$-ciphertext (by induction on i). To solve for $c_1, \ldots c_{\mathsf{sec}/2}$, we consider the *lowest* index j such that $\mathbf{e}_j \neq \mathbf{e}'_j$, construct an lower triangular matrix in a similar way as above, and solve from the first equation downwards. We conclude that $\mathbf{c}$ contains $(N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{(1/2+\nu)\mathsf{sec}}, d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{(1/2+\nu)\mathsf{sec}})$-ciphertexts.

We note that since the verifier accepted, each $\mathbf{z}_i$ has small norm and decodes to a value in $(\mathbb{F}_{p^k})^s$. Since we can write $\mathbf{x}_i$ as a linear combination of the $\mathbf{z}_i$, it follows from correctness of the

cryptosystem that the $\mathbf{x}_i$ also decode to values in $(\mathbb{F}_{p^k})^s$. Finally, if diag was set to true, the verifier only accepts if all $\mathbf{z}_i$ decode to diagonal values. Again, since we can write $\mathbf{x}_i$ as a linear combination of the $\mathbf{z}_i$, the $\mathbf{x}_i$ also decode to diagonal values.

Zero-Knowledge: We give an honest-verifier simulator for the protocol that outputs accepting conversations. In order to simulate one repetition, the simulator samples $\mathbf{e} \in \{0,1\}^{\mathsf{sec}}$ uniformly and $\mathbf{z}, T$ uniformly with the constrain that $\mathbf{d}$ contains random ciphertexts satisfying the verifiers check, i.e., $\mathbf{z}_i, \mathbf{t}_i$ are uniform, subject to $\|\mathbf{z}_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$, $\|\mathbf{t}_i\|_\infty \leq d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$, where moreover $\mathbf{z}_i$ is generated as $\mathsf{encode}(\mathbf{m}_i)+\mathbf{u}_i$ where $\mathbf{m}_i$ is a random plaintext (diagonal if diag is set to true) and $\mathbf{u}_i$ contains multiples of p that are uniformly random, subject to $\|\mathbf{z}_i\|_\infty \leq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\nu\mathsf{sec}-1}$. Finally, $\mathbf{a}$ is computed as $\mathbf{a}^\mathsf{T} \leftarrow \mathbf{d}^\mathsf{T} \boxminus (M_\mathbf{e} \cdot \mathbf{c}^\mathsf{T})$. In the real conversation, the provers choice of values in $\mathbf{z}_i$ and $\mathbf{t}_i$ are statistically close to the distribution used by the simulator. This is because the prover uses the same method to generate these values, except that he adds in some vectors of exponentially smaller norm which leads to a statistically close distribution. Since $\mathbf{e}$ has the correct distribution and $\mathbf{a}$ follows deterministically from the last two messages, the simulation is statistically indistinguishable.

□

We now give a protocol that leads to smaller values of the parameters and hence also allows better parameters for the underlying cryptoystem. This version, however, is better suited for use with the Fiat-Shamir heuristic. The idea is to let the prover choose his randomness in a smaller interval, and abort if the last message would reveal too much information. This is an idea from [21]. When using the Fiat-Shamir heuristic, this is not a problem as he prover only needs to show a successful attempt to he verifier. We let h be a suitable hash function that outputs sec-bit strings.

Protocol Π_{ZKPoPK}

- For $i = 1, \ldots, V$, the prover generates $\mathbf{y}_i \leftarrow \mathbb{Z}^N$ and $\mathbf{s}_i \leftarrow \mathbb{Z}^d$, such that $\|\mathbf{y}_i\|_\infty \leq 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2$ and $\|\mathbf{s}_i\|_\infty \leq 128 \cdot d \cdot \rho \cdot \mathsf{sec}^2$. For $\mathbf{y}_i$, this is done as follows: choose a random message $\mathbf{m}_i \in (\mathbb{F}_{p^k})^s$ and set $\mathbf{y}_i = \mathsf{encode}(\mathbf{m}_i) + \mathbf{u}_i$, where each entry in $\mathbf{u}_i$ is a multiple of p, chosen uniformly at random, subject to $\|\mathbf{y}_i\|_\infty \leq 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2$. If diag is set to true then the $\mathbf{m}_i$ are additionally chosen to be diagonal elements.
- The prover computes $a_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i)$, for $i = 1, \ldots, V$, and defines $S \in \mathbb{Z}^{V \times d}$ to be the matrix whose ith row is $\mathbf{s}_i$ and sets $\mathbf{y} \leftarrow (\mathbf{y}_1, \ldots, \mathbf{y}_V)$, $\mathbf{a} \leftarrow (a_1, \ldots, a_V)$.
- The prover sends $\mathbf{a}$ to the verifier.
- The prover computes $\mathbf{e} = h(\mathbf{a}, \mathbf{c})$.
- The prover sets $\mathbf{z} \leftarrow (\mathbf{z}_1, \ldots, \mathbf{z}_V)$, such that $\mathbf{z}^\mathsf{T} = \mathbf{y}^\mathsf{T} + M_\mathbf{e} \cdot \mathbf{x}^\mathsf{T}$, and $T = S + M_e \cdot R$. Let $\mathbf{t}_i$ be the ith row of T. If for any i, it is the case that $\|\mathbf{z}_i\|_\infty > 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2 - \tau \cdot \mathsf{sec}$ or $\|\mathbf{t}_i\|_\infty > 128 \cdot d \cdot \rho \cdot \mathsf{sec}^2 - \rho \cdot \mathsf{sec}$, the prover aborts and the protocol is restarted. Otherwise the prover sends $(\mathbf{a}, \mathbf{z}, T)$ to the verifier.
- The verifier computes $\mathbf{e} = h(\mathbf{a}, \mathbf{c}), d_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i)$, for $i = 1, \ldots, V$, where $\mathbf{t}_i$ is the ith row of T and sets $\mathbf{d} \leftarrow (d_1, \ldots, d_V)$.
- The verifier checks $\mathsf{decode}(\mathbf{z}_i) \in \mathbb{F}_{p^k}^s$ and whether the following three conditions hold

$$\mathbf{d}^\mathsf{T} = \mathbf{a}^\mathsf{T} \boxplus \left(M_\mathbf{e} \boxtimes \mathbf{c}^\mathsf{T}\right), \quad \|\mathbf{z}_i\|_\infty \leq 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2, \quad \|\mathbf{t}_i\|_\infty \leq 128 \cdot d \cdot \rho \cdot \mathsf{sec}^2.$$

If diag is set to true the verifier also checks whether $\mathsf{decode}(\mathbf{z}_i)$ is a diagonal element, and rejects if it is not.

Fig. 10. The ZKPoPK Protocol, version for Fiat-Shamir heuristic.

We claim that the Fiat-Shamir based protocol is a proof of knowledge for the relation in question in the random oracle model. In this case, however, we can guarantee that the adversarially generated ciphertexts are $(N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8}, d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8})$- ciphertexts.

Completeness: Assume the prover is honest. Note first that each $\mathbf{y}_i$ was constructed to be congruent mod p to the encoding of a value in $(\mathbb{F}_{p^k})^s$. Since this is also the case for the $\mathbf{x}_i$'s if the prover is honest, the same is true for the $\mathbf{z}_i$'s, and they therefore always decode to a value in $(\mathbb{F}_{p^k})^s$. If diag was set to true, all $\mathbf{x}_i, \mathbf{y}_i$ contain diagonal plaintexts, and then the same is true for the $\mathbf{z}_i$.

Next, for $i = 1, \ldots, V$ the verifier checks if $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i)$ equals $a_i \boxplus M_{\mathbf{e},i} \cdot \mathbf{c}^\mathsf{T}$, since $M_{\mathbf{e},i}$ is a scalar matrix we write multiplication with $\cdot$ as opposed to $\boxtimes$. The check passes because of the following relation:

$$\begin{aligned}
a_i \boxplus \left(M_{\mathbf{e},i} \cdot \mathbf{c}^\mathsf{T}\right) &= \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i) \boxplus_{k=1}^{\mathsf{sec}} \left(M_{\mathbf{e},i,k} \cdot c_k\right) \\
&= \mathsf{Enc}_{\mathsf{pk}}(\mathbf{y}_i, \mathbf{s}_i) \boxplus_{k=1}^{\mathsf{sec}} \left(M_{\mathbf{e},i,k} \cdot \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_k, \mathbf{r}_k)\right) \\
&= \mathsf{Enc}_{\mathsf{pk}}\left(\mathbf{y}_i + \sum_{k=1}^{\mathsf{sec}} M_{\mathbf{e},i,k} \cdot \mathbf{x}_k, \quad \mathbf{s}_i + \sum_{k=1}^{\mathsf{sec}} M_{\mathbf{e},i,k} \cdot \mathbf{r}_k\right) \\
&= \mathsf{Enc}_{\mathsf{pk}}\left(\mathbf{y}_i + M_{\mathbf{e},i} \cdot \mathbf{x}^\mathsf{T}, \quad \mathbf{s}_i + M_{\mathbf{e},i} \cdot \mathbf{r}^\mathsf{T}\right) = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i).
\end{aligned}$$

Moreover, given that $\mathbf{z}_i = \mathbf{y}_i + M_{\mathbf{e},i} \cdot \mathbf{x}^\mathsf{T}$ and that all ciphertexts in $\mathbf{c}$ are (τ, ρ)-ciphertexts, we get that each single coordinate in $M_{\mathbf{e},i} \cdot \mathbf{x}^\mathsf{T}$ is numerically at most $\mathsf{sec} \cdot \tau$. Each coordinate of $\mathbf{y}_i$ was chosen from an interval that is a factor $128 \cdot N \cdot \mathsf{sec}$ larger. Therefore each coordinate in $\mathbf{z}_i$ fails to be in the required range with probability $1/(128 \cdot N \cdot \mathsf{sec})$. Note that this probability does not depend on the concrete values of the coordinates in $M_{\mathbf{e},i} \cdot \mathbf{x}^\mathsf{T}$, only on the bound on the numeric value.

By a union bound over the N coordinates of $\mathbf{z}_i$ we get that $\|\mathbf{z}_i\|_\infty \leq 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2 - \tau \cdot \mathsf{sec}$ fails with probability at most $1/(128 \cdot \mathsf{sec})$, and by a final union bound over the $2\,\mathsf{sec} - 1$ ciphtertexts that all checks on the $\mathbf{z}_i$'s are ok except with probability at most $1/64$. A similar argument shows that the check $\|\mathbf{t}_i\|_\infty \leq 128 \cdot d \cdot \rho \cdot \mathsf{sec}^2 - \rho \cdot \mathsf{sec}$ fails also with probability at most $1/64$. The conclusion is that the prover will abort with probability at most $1/32$, so we expect to only have to repeat the protocol once to have success.

Soundness: By a standard argument, a prover who can efficiently produce a valid proof is able to produce $(x, \mathbf{a}, \mathbf{e}, (\mathbf{z}, T))$ and $(x, \mathbf{a}, \mathbf{e}', (\mathbf{z}', T'))$ with $\mathbf{e} \neq \mathbf{e}'$ that the verifier would accept. Since both checks $\mathbf{d}^\mathsf{T} = \mathbf{a}^\mathsf{T} \boxplus (M_{\mathbf{e}} \cdot \mathbf{c}^\mathsf{T})$ and $\mathbf{d}'^\mathsf{T} = \mathbf{a}^\mathsf{T} \boxplus (M_{\mathbf{e}'} \cdot \mathbf{c}^\mathsf{T})$ passed, one can subtract the two equalities and obtain

$$(M_{\mathbf{e}} - M_{\mathbf{e}'}) \boxtimes \mathbf{c}^\mathsf{T} = \left(\mathbf{d} \boxminus \mathbf{d}'\right)^\mathsf{T} \tag{2}$$

In order to find $\mathbf{x}$ and R such that $c_k = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_k, \mathbf{r}_k)$ for $k = 1, \ldots, \mathsf{sec}$, we first solve (2) as a linear system in $\mathbf{c}$. Let j be the highest index such that $\mathbf{e}_j \neq \mathbf{e}'_j$. The $\mathsf{sec} \times \mathsf{sec}$ submatrix of $M_{\mathbf{e}} - M_{\mathbf{e}'}$, consisting of the rows of $M_{\mathbf{e}} - M_{\mathbf{e}'}$ between j and $j + \mathsf{sec} - 1$ both included, is upper triangular with entries in $\{-1, 0, 1\}$ and its diagonal consists of the non-zero value $\mathbf{e}_j - \mathbf{e}'_j$ (so it is possible to find a solution for $\mathbf{c}$). Since the verifier has values $\mathbf{z}_i, \mathbf{t}_i, \mathbf{z}'_i, \mathbf{t}'_i$ such that $d_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}_i, \mathbf{t}_i)$ and $d'_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}'_i, \mathbf{t}'_i)$, and given that $c_i = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}_i, \mathbf{r}_i)$, it is possible to directly solve the linear system in $\mathbf{x}$ and R (since the cryptosystem is additively homomorphic), from the bottom equation to the one "in the middle" with index $\mathsf{sec}/2$.

Since $\|\mathbf{z}_i\|_\infty, \|\mathbf{z}'_i\|_\infty \leq 128 \cdot N \cdot \tau \cdot \mathsf{sec}^2$ and $\|\mathbf{t}_i\|_\infty, \|\mathbf{t}'_i\|_\infty \leq 128 \cdot d \cdot \rho \cdot \mathsf{sec}^2$, we conclude that $c_{\mathsf{sec}-i}$ must be a $(256 \cdot N \cdot \tau \cdot 2^i \cdot \mathsf{sec}^2, 256 \cdot d \cdot \rho \cdot 2^i \cdot \mathsf{sec}^2)$-ciphertext (by induction on i). To solve for $c_1, \ldots c_{\mathsf{sec}/2}$, we consider the *lowest* index j such that $\mathbf{e}_j \neq \mathbf{e}'_j$, construct an lower triangular matrix in a similar way as above, and solve from the first equation downwards. We conclude that $\mathbf{c}$ contains $(N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{sec/2+8}, d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{sec/2+8})$-ciphertexts.

We note that since the verifier accepted, each $\mathbf{z}_i$ has small norm and decodes to a value in $(\mathbb{F}_{p^k})^s$. Since we can write $\mathbf{x}_i$ as a linear combination of the $\mathbf{z}_i$, it follows from correctness of the cryptosystem that the $\mathbf{x}_i$ also decode to values $(\mathbb{F}_{p^k})^s$. Finally, if diag was set to true, the verifier only accepts if all $\mathbf{z}_i$ decode to diagonal values. Again, since we can write $\mathbf{x}_i$ as a linear combination of the $\mathbf{z}_i$, the $\mathbf{x}_i$ also decode to diagonal values.

Zero-Knowledge: We give an honest-verifier simulator for the protocol that outputs an accepting conversation (that does not abort).

In order to simulate one repetition, the simulator samples $\mathbf{e} \in \{0,1\}^{\mathsf{sec}}$ uniformly and $\mathbf{z}, T$ uniformly with the constrain that $\mathbf{d}$ contains random $(8 \cdot N \cdot \tau \cdot \mathsf{sec}^2 - \tau \cdot \mathsf{sec}, 8 \cdot d \cdot \rho \cdot \mathsf{sec}^2 - \rho \cdot \mathsf{sec})$-ciphertexts. where moreover $\mathbf{z}_i$ is generated as $\mathsf{encode}(\mathbf{m}_i) + \mathbf{u}_i$ where $\mathbf{m}_i$ is a random plaintext (a diagonal one if diag is set to true) and $\mathbf{u}_i$ contains multiples of p that are uniformly random, subject to $\|\mathbf{z}_i\|_\infty \leq 8N \cdot \tau \cdot \mathsf{sec}^2 - \tau \cdot \mathsf{sec}$. Finally, $\mathbf{a}$ is computed as $\mathbf{a}^\mathsf{T} \leftarrow \mathbf{d}^\mathsf{T} \boxminus (M_\mathbf{e} \cdot \mathbf{c}^\mathsf{T})$. Define the random oracle to output $\mathbf{e}$ on input $\mathbf{a}, \mathbf{c}$, output $(\mathbf{a}, \mathbf{e}, (\mathbf{z}, T))$ and stop.

We argue that this simulation is perfect: The distribution of a simulated $\mathbf{e}$ is the same as a real one. Also, it is straightforward to see that in a real conversation, given that the prover does not abort, the vectors $\mathbf{z}_i, \mathbf{t}_i$ will be uniformly random, subject to $\|\mathbf{z}_i\|_\infty \leq 8 \cdot s \cdot \tau \cdot \mathsf{sec}^2 - \tau \cdot \mathsf{sec}$ and $\|\mathbf{t}_i\|_\infty \leq 8 \cdot d \cdot \rho \cdot \mathsf{sec}^2 - \rho \cdot \mathsf{sec}$. So the simulator chooses $\mathbf{z}_i, \mathbf{t}_i$ with exactly the right distribution. Since the value of $\mathbf{a}$ follows deterministically from the $\mathbf{e}, \mathbf{z}_i, \mathbf{t}_i$, we have what we wanted.

Doing without random oracles. The above protocol can also be executed without using the Fiat-Shamir heuristic. In this case, the prover will start $\mathsf{sec}/5$ instances of the protocol, computing $\mathbf{a}_1, \ldots, \mathbf{a}_{\mathsf{sec}/5}$. We choose this number of instance because it will ensure that the prover fails on all of them with probability only $(1/32)^{\mathsf{sec}/5} = 2^{-\mathsf{sec}}$. The prover commits to all these values, which can be done, for instance, with a Merkle hash tree, in which case the commitment will be very short, and any of $\mathbf{a}$'s can be opened by sending a piece of information that is only logarithmic in sec.

The verifier selects $\mathbf{e}$, the prover finds an instance where he would not abort the protocol with this $\mathbf{e}$, opens the corresponding $\mathbf{a}$ and completes that instance.

This is complete and zero-knowledge by the same argument as above plus the hiding property of the commitment scheme used. Soundness follows from the fact that if the prover succeeds with probability significantly greater that $2^{-\mathsf{sec}} \cdot \mathsf{sec}/5$ he must be able to answer different challenges correctly for some fixed instance out of the $\mathsf{sec}/5$ we have. Such answers can be extracted by rewinding, and then the rest of the argument is the same as above.

A.2 The UC Model

In the following sections, we show that the online and preprocessing phases of our protocol are secure in the UC model. We briefly recall how this model works: we will use the variant where there is only one adversarial entity, the environment $\mathcal{Z}$. The environment chooses inputs for the honest players and gets their outputs when the protocol is done. It also does an attack on the protocol

which is our case means that it corrupts up to $n-1$ of the players and takes control over their actions. When $\mathcal{Z}$ stops, it outputs a bit. This process where $\mathcal{Z}$ interacts with the real players and protocol is called the real process.

To define what it means that the protocol implements functionality $\mathcal{F}$ securely we assume there exists a simulator $\mathcal{S}$ that interacts with both $\mathcal{F}$ and $\mathcal{Z}$. Towards $\mathcal{F}$, it chooses inputs for the corrupt players and will get their outputs. Towards $\mathcal{Z}$, it must simulate a view of the protocol that looks like what $\mathcal{Z}$ would see in a real attack. This process is called the ideal process, and here $\mathcal{F}$ supplies $\mathcal{Z}$ with the i/o interface of honest players. We say that the protocol implements $\mathcal{F}$ securely if $\mathcal{Z}$ outputs 1 with essentially the same probability in the real as in the ideal process. We speak of computational security if $\mathcal{Z}$ is assumed to be poly-time bounded and of statistical security if $\mathcal{Z}$ is unbounded.

A.3 Online Phase

On generating the e_i's Before proving the online protocol UC secure, we compute the probability of getting away with cheating in step 4 of 'Output' and how this depends on the way we generate the e_i's.

For this purpose we design the following security game:

1. The challenger generates the secret key α and MACs $\gamma_i \leftarrow \alpha m_i$ and sends messages $m_1, \ldots, m_T$ to the adversary.
2. The adversary sends back messages $m'_1, \ldots, m'_T$.
3. The challenger generates random values $e_1, \ldots, e_T \leftarrow \mathbb{F}_{p^k}$ and sends them to the adversary.
4. The adversary provides an error Δ.
5. Set $m \leftarrow \sum_{i=0}^{T} e_i m'_i, \gamma \leftarrow \sum_{i=0}^{T} e_i \gamma_i$. Now, the challenger checks that $\alpha m = \gamma + \Delta$

The adversary wins the game if there is an i for which $m'_i \neq m_i$ and the final check goes through.

It is not difficult to see that this game indeed models 'Output'(up to step 4): The second step in the game where the adversary sends the m'_i's models the fact that corrupted players can choose to lie about their shares of values opened during the protocol execution. Δ models the fact that the adversary is allowed to introduce errors on the macs when data are sent to $\mathcal{F}_{\text{PREP}}$ in the initial part of the protocol and may also modify the shares of macs held by corrupt players. Finally, since α, γ are secret shared in the protocol, the adversary has no information on α, γ ahead of time in the protocol, just as in the security game.

Now, let us look at the probability of winning the game if the e_i's are randomly chosen. If the check goes through, we have that the following equality holds: $\alpha \sum_{i=0}^{T} e_i(m'_i - m_i) = \Delta$. First we consider the case where $\sum_{i=0}^{T} e_i(m'_i - m_i) \neq 0$, so $\alpha = \Delta / \sum_{i=0}^{T} e_i(m'_i - m_i)$. This implies that being able to pass the check is equivalent to guessing α. However, since the adversary has no information about α, this happens with probability only $1/|\mathbb{F}_{p^k}|$. So what is left is to argue that $\sum_{i=0}^{T} e_i(m'_i - m_i) = 0$ also happens with very low probability. This can be seen as follows. We define $\mu_i := (m'_i - m_i)$ and $\mu := (\mu_1, \ldots, \mu_T), e := (e_1, \ldots, e_T)$. Now $f_\mu(e) := e \cdot \mu = \sum_{i=0}^{T} e_i \mu_i$ defines a linear mapping, which is not the 0-mapping since at least one $\mu_i \neq 0$. From linear algebra we then have the rank-nullity theorem telling us that $\dim(\ker(f_\mu)) = T - 1$. Also since e is random and the adversary does not know e when choosing the m'_i's, the probability of $e \in \ker(f_\mu)$ is $|\mathbb{F}_{p^k}^{T-1}|/|\mathbb{F}_{p^k}^{T}| = 1/|\mathbb{F}_{p^k}|$. Summing up, the total probability of winning the game is at most $2/|\mathbb{F}_{p^k}|$.

Since choosing the e_i's uniformly would require an expensive coin-flip protocol, we use a different way to generate them in the protocol: namely e_1 is chosen at random and for $i > 1$, $e_i \leftarrow e_1^i$.

This has the advantage of adding only a constant number of multiplications in $\mathbb{F}_{p^k}$ for a secure multiplication. On the security side, we still want that $\sum_{i=0}^{T} e_i \mu_i = 0$ should happen with small probability. Viewing f_μ as a polynomial of degree T, we know it has at most T roots, so we have to make sure we have an upper bound on T such that e_1 is chosen from a field big enough for T/p^k to be negligible.

An alternative approach would be to use a pseudorandom generator G. We would then have shared some random seed $\langle s \rangle$. By opening $\langle s \rangle$ and feeding it to G we can generate T pseudorandom elements. In the protocol, the parties would commit to their share of the MAC on s, and when α becomes public, the MAC would be checked. If it is OK, the protocol would go on with the rest of the checks. With respect to cheating the argument is basically the same; If an adversary A has a significant probability of choosing m_i''s such that $\sum_{i=0}^{T} e_i(m_i' - m_i) = 0$, then the G is a bad pseudorandom generator, or in other words, we can use A to break G. With this way of generating the e_i's, we increase the complexity for one secure multiplication by whatever G needs to generate one pseudorandom element.

Proof (Theorem 1). We construct a simulator $\mathcal{S}_{\text{AMPC}}$ such that a poly-time environment $\mathcal{Z}$ cannot distinguish between the real protocol system and the ideal. We assume here static, active corruption. The simulator runs a copy of the protocol Π_{ONLINE} and simulates the ideal functionalities for preprocessing and commitment. It relays messages between parties/$\mathcal{F}_{\text{PREP}}$ and $\mathcal{Z}$, such that $\mathcal{Z}$ will see the same interface as when interacting with a real protocol. The specification of the simulator $\mathcal{S}_{\text{AMPC}}$ is presented in Figure 11.

Simulator $\mathcal{S}_{\text{AMPC}}$

Initialize: The simulator creates the desired number of triples by doing the steps in $\mathcal{F}_{\text{PREP}}$. Note that here the simulator will read all data of the corrupted parties specified to the copy of $\mathcal{F}_{\text{PREP}}$.

Rand: The simulator runs the copy protocol honestly and calls *rand* on the ideal functionality $\mathcal{F}_{\text{AMPC}}$.

Input: If P_i is not corrupted the copy is run honestly with dummy input, for example 0.
If P_i is corrupted the input step is done honestly and then the simulator waits for P_i to broadcast δ. Given this, the simulator can compute $x_i' \leftarrow (r + \delta)$ since it knows (all the shares of) r. This is the supposed input of P_i, which the simulator now gives to the ideal functionality $\mathcal{F}_{\text{AMPC}}$.

Add: The simulator runs the protocol honestly and calls *add* on the ideal functionality $\mathcal{F}_{\text{AMPC}}$.

Multiply: The simulator runs the protocol honestly and calls *multiply* on the ideal functionality $\mathcal{F}_{\text{AMPC}}$.

Output: The output step is run and the protocol is aborted if one of the checks in step 4 does not go through. Otherwise the simulator calls *output* on $\mathcal{F}_{\text{AMPC}}$ and gets the result y back. Now it has to simulate shares y_j of honest parties such that they are consistent with y. Note that the simulator already has shares of an output value y' that was computed using the dummy inputs, as well as shares of the MAC for y'. The simulator now selects an honest party, say P_k and adds $y - y'$ to his share of y and $\alpha(y - y')$ to his share of the MAC. Note that the simulator can compute $\alpha(y - y')$ since it knows from the beginning (all the shares of) α. Now it simulates the openings of shares of y towards the environment according to the protocol. If this terminates correctly, send OK to $\mathcal{F}_{\text{AMPC}}$ (causing it to output y to the honest players).

Fig. 11. The simulator for $\mathcal{F}_{\text{AMPC}}$.

To see that the simulated and real processes cannot be distinguished, we will show that the view of the environment in the ideal process is statistically indistinguishable from the view in the real process. This view consists of the corrupt players' view of the protocol execution as well as the inputs and outputs of honest players.

We first argue that the view up to the point where the output value is opened (step 5 of the 'output' stage of the protocol) has exactly the same distribution in the real and in the simulated

case: First, the value broadcast by honest players in the input stage are always uniformly random. Second, when a value is partially opened in a secure multiplication, fresh shares of a random value are subtracted, so the honest players will always send a set of uniformly random and independent values. Third, the honest players hold shares in MACs on the opened values, these are random sharings of a correct MAC with an error added that is determined by the errors specified by the environment in the initial phase. Therefore, also the MAC and shares revealed in step 4 of 'output' have the same distribution in the simulated as in the the real process. Finally note that if the simulated protocol aborts, the simulator makes the ideal functionality fail, so the environment will see that honest players generate no output, just as when the real process aborts.

Now, if the real or simulated protocol proceeds to the last step, the only new data that the environment sees is an output value y, plus some shares of honest players. These are random shares that are consistent with y and its MAC in both the simulated and real case. In other words, the environments' view of the last step has the same distribution in real and simulated case as long as y is the same.

In the simulation, y is of course the correct evaluation on the inputs matching the shares that were read from the corrupted parties in the beginning. To finish the proof, it is therefore sufficient to show that the same happens in the real process with overwhelming probability. In other words, the event that the real protocol terminates but the output is not correct occurs with negligible probability.

Incorrect outputs can result either from corrupted parties who during the protocol successfully cheat with their shares or from having computed with triples where the multiplicative relation does not hold (even if the revealed shares were correct). For the latter case we argue that with correct shares the multiplicative relation holds with overwhelming probability, and this follows from the check on the triples in step 1 of 'Multiply': It is easy to see that if the triples are correct, the check will be true. On the other hand, if some triple is not correct, (in spite of correct shares), the probability of satisfying the check is $1/|\mathbb{F}_{p^k}|$, since there is only one random challenge t, for which $t \cdot (c - a \cdot b) = (h - g \cdot f)$. For the former case regarding the checking of shares, we have checks related to the openings of $[\![\cdot]\!]$-values (during 'Input and a single one in 'Output'). The rest of the checking is done in steps 4 and 5 of 'Output'. Being able to cheat during an opening of a $[\![\cdot]\!]$-value corresponds to guessing at least one private key β_i. Assuming β_i is chosen randomly in $\mathbb{F}_{p^k}$, the probability is at most $1/|\mathbb{F}_{p^k}|$. Furthermore, as we discussed in the beginning of this section, the probability of a party being able to cheat in step 4 is $(T+1)/|\mathbb{F}_{p^k}|$ where T is the number of values opened during secure multiplications. In step 5, only one MAC is checked for each output, so here the probability of cheating is $1/|\mathbb{F}_{p^k}|$ per check as argued earlier. Since the protocol aborts as soon as a check fails, the probability that it terminates with an incorrect output is the maximum probability with which any single check can be cheated, which in our case is $(T+1)/|\mathbb{F}_{p^k}|$. This is negligible, since we assume that T is polynomial while p^k is exponential in sec. □

Commitments based on $\mathcal{F}_{\text{PREP}}$ In the above we assumed access to an ideal functionality for commitments. We can, however, do the commitments needed in our protocol based only on the output of $\mathcal{F}_{\text{PREP}}$ as follows. First a random value $[\![r]\!]$ is opened to the committer P_i (This could even be done in the preprocessing). To commit to a value x, P_i broadcasts $c = r + x$. To open the commitment, $[\![r]\!]$ is opened to all the players who can now compute $c - r = x$. Correctness is still guaranteed because of the MACs in $[\![r]\!]$. Furthermore, since to begin with $[\![r]\!]$ is only opened to P_i, we have that c is indistinguishable from a random value and can thus easily be simulated. To simulate during 'Output' when P_i is honest and has to open his commitment, the simulator simply changes P_i's

share of $[\![r]\!]$ and the shares of the MACs to make it fit with the broadcasted value and the value he should have committed to. This is possible because the simulator knows all MAC keys. It is easy to see that this has communication and computational complexity $O(n^2)$ per commitment.

Implementing Broadcast and Multiple Inputs/Outputs To implement broadcast based on point-to-point channels, we first observe that since we do not guarantee termination anyway, the broadcast does not have to terminate either. Therefore the following very simple protocol for broadcasting $x \in \mathbb{F}_{p^k}$ is sufficient:

1. The broadcaster sends x to all players.
2. Each player sends to all players what he received in the previous step.
3. Each player checks that he received the value x from all players. If, so output x, otherwise abort.

This protocol has communication complexity $O(n^2)$ field elements for one broadcast. However, this can be optimized in case we need to broadcast many values. Below, we assume each player sends one value, say P_i wants to send x_i. We also assume that we have a random value $[\![s]\!]$ from the preprocessing, and that we have an ϵ-almost universal class of hash functions $\{h_s\}$ for negligible ϵ, indexed by values s, taking as input strings of n elements in $\mathbb{F}_{p^k}$ and producing output in $\mathbb{F}_{p^k}$. A simple example is where we view the input F as specifying coefficients of a polynomial of degree $n-1$, and $h_s(F)$ is the result of evaluating this polynomial in point s. If two inputs F, F' are distinct, their difference has at most $n-1$ roots, so the probability that $h_s(F) = h_s(F')$ is $(n-1)/p^k$. The protocol goes as follows:

1. P_i sends x_i to all players.
2. $[\![s]\!]$ is opened.
3. Each player sends to all players $h_s(F)$ where F is the string of values he received in the first step.
4. Each players checks that he received the same hash value from all players. If, so output $x_1, \ldots, x_n$ as received in the first step, otherwise abort.

It is clear that if a player sent different data to different honest players, some honest player will abort, except with probability $(n-1)/p^k$. This protocol has complexity $O(n^2)$, including also the cost of opening $[\![s]\!]$. But the cost per value we broadcast is only $O(n)$. This protocol generalizes easily to a case where one player has n values to broadcast.

In the online protocol we specified before, broadcast is used to give inputs in the first stage. Here, all players broadcast a value, and this is readily implemented with the optimized broadcast protocol above, so we get complexity $O(n)$ per input gate. If players have several inputs, we just execute several instances of this broadcast.

The only other point where broadcast is used is in partial openings where a designated player P_1 broadcasts the value that is be to opened. Here, we can simply buffer the values sent until we have n of them and then do the check in step 3-4 above that P_1 has sent the same values to all players. Note that even if we allow P_1 to send different data to different players for a while, this does not allow information to leak: the fact observed in the simulation proof above, that in any partial opening the honest players always send random independent values, still holds even if P_1 has sent inconsistent data in previous rounds.

A.4 Running the online Phase with Small Fields

Suppose we want error probability $2^{-\mathsf{sec}}$, and $\log p^k$ is much smaller than sec.

When we consider how to solve this problem, we will at first ignore Step 1 in the **Multiply** stage on the online protocol, where one triple is "sacrificed" to check another, as this step could be done as part of the preprocessing. Nevertheless we do not want to ignore the fact that this step will have a large error probability $1/p^k$. We could solve this by sacrificing $D = \lceil \frac{\mathsf{sec}}{\log p^k} \rceil$ triples instead of one, but we can do much better, and this is described below in Section "A smaller sacrifice" below.

Going back to the actual online phase, we can compensate for the fact that $\log p^k$ is much smaller than sec by setting up the preprocessing so it can work over an extension field K of $\mathbb{F}_{p^k}$ of degree $D = \lceil \frac{\mathsf{sec}}{\log p^k} \rceil$, i.e. an element in K is represented as $\lceil \frac{\mathsf{sec}}{\log p^k} \rceil$ elements from $\mathbb{F}_{p^k}$. All MAC keys and MACs will be generated in K whereas all values to be computed on will still be in $\mathbb{F}_{p^k}$. The preprocessing can ensure this because the ZK proof can already force a prover to choose plaintexts that decode to elements in a subfield of K.

Then error probabilities in the proof of the online phase that were $1/p^k$ before will now be $1/|K| \leq 2^{-\mathsf{sec}}$. The computational complexity of the online phase will now be $O(n|C| + n^3)$ elementary operations in K. Asymptotically, this amounts to $O((n|C| + n^3)D \log D \log\log D)$ elementary operations in $\mathbb{F}_{p^k}$, where the overhead for storage and communication is just D.

It is also possible to get error probability $2^{-\mathsf{sec}}$ while having the preprocessing work only over $\mathbb{F}_{p^k}$. Here the overhead will be larger namely $D^2 \log D \log\log D$, but this may be the best option when D is not very large. The idea is to authenticate by doing D MACs in parallel over $\mathbb{F}_{p^k}$ for every authenticated value, using D independent keys.

We will still do the linear combination $a = \sum_j e_j a_j$ over K, where $e_j = e^j$. This can be done by having the preprocessing generate D random values and thinking of these as an element $e \in K$. Note, however, that we also have to compute a linear combination of the corresponding shares of MACs, i.e., $\gamma_i = \sum_j e_j \gamma(a_j)_i$, and we have D such MACs in parallel. This is why we get a overhead factor $D^2 \log D \log\log D$ for the computational work in this case.

A Smaller Sacrifice. In this section we describe a different method to check the multiplicative relation on triples $\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$, where $a, b, c \in \mathbb{F}_{p^k}$. The aim is to decrease the (amortized) number of triples to sacrifice per check. Our approach resembles a technique introduced by Ben-Sasson et al in [4] and one by Cramer et al in [10].

The first step in our construction is to consider a batch of $t+1$ triples $\langle a_i \rangle, \langle b_i \rangle, \langle c_i \rangle$ for $i = 1, \ldots, t+1$ at once. There are two main ideas in the construction: the first one is to interpolate the values and get polynomials $A, B, C \in \mathbb{F}_{p^k}[X]$ such that $A(i) = a_i$, $B(i) = b_i$, $C(i) = c_i$; if the triples where correctly generated, one would expect $A(x)B(x) = C(x)$ for all x. The second idea is to think of A, B, C as polynomials over a field extension K of $\mathbb{F}_{p^k}$, so that one can check the expected multiplicative relation evaluating A, B, C at a random element $z \in K$; the probability that the check passes even if some of the triples did not satisfy the relation is inversely proportional to the size of K. We now present the full construction.

- Let $\langle a_i \rangle, \langle b_i \rangle, \langle c_i \rangle$, $i = 1, \ldots, t+1$, be a batch of triples to check.
- One can think of the values $a_1, \ldots, a_{t+1}$ (resp. $b_1, \ldots, b_{t+1}$) as $t+1$ evaluations over $\mathbb{F}_{p^k}$ of a unique polynomial $A \in \mathbb{F}_{p^k}[X]$ (resp. $B \in \mathbb{F}_{p^k}[X]$) of degree t. Concretely, one can define the polynomial A (resp. B) such that $A(i) = a_i$ (resp. $B(i) = b_i$). Since the coefficients of A (resp.

B) can be computed as a linear combination of the a_i's (resp. b_i's), the players can compute representations of such coefficients by local computation.
- Players can compute $\langle a_{t+2}\rangle, \ldots, \langle a_{2t+1}\rangle$ such that $A(i) = a_i$, again by local computation, since evaluating a polynomial is a linear operation.
- Players can engage in the multiplication step of the online phase with input $\langle a_i\rangle, \langle b_i\rangle$, and get $\langle c_i\rangle$ (hopefully $c_i = a_i b_i$) for $i = t+2, \ldots, 2t+1$. Notice that players call the multiplication step t times here, so they sacrifice t triples.
- Using only linear computation players can now compute representations of coefficients of the unique polynomial $C \in \mathbb{F}_{p^k}[X]$ of degree $2t$ such that $C(i) = c_i$ for $i = 1, \ldots, 2t+1$.
- Let K be a field extension of $\mathbb{F}_{p^k}$ of degree D. It is possible to think of A, B, C as polynomials over K, by embedding the coefficients via the natural map $\mathbb{F}_{p^k} \longrightarrow K$. Players now evaluate representations for $A(z)B(z)$, and $C(z)$, where z is a public random element in K, and check if $A(z)B(z) = C(z)$ by outputting $A(z)B(z) - C(z)$ and checking if the result is zero. This check can be repeated a number of times in order to lower the error probability. If the check passed all the times, players consider the original triples as valid; otherwise, they discard the triples and start again with fresh triples.

Notice that in order to compute $A(z)B(z)$ and $C(z)$, players need to compute at most D^2 multiplications over $\mathbb{F}_{p^k}$, since $A(z)B(z)$ can be computed by multiplying a $D \times D$ matrix (dependent of $A(z)$) with the vector $B(z)$ (over K, multiplication by a fixed element is an endomorphism of K as a $\mathbb{F}_{p^k}$-vector space). Notice also that we may use the old method of sacrificing more than one triple per multiplication to get any desired error probability for the multiplications over $\mathbb{F}_{p^k}$. We analyze below the error probability we must require.

For the analysis of the construction, one sees that if the multiplicative relation was satisfied by all the original triples, the polynomials AB and C are equal, so the final test passes. In case the triples did not satisfy the relation, then the polynomials AB and C are different, but since they are both of degree at most $2t$, they can agree in at most $2t$ points. Therefore, if z is a root of $AB - C$, then the test passes, and uniform elements in K are roots of $AB - C$ with probability at most $2t/|K|$. If z is not a root of $AB - C$, the test passes only if the multiplication $A(z)B(z)$ does give the correct result, so if we make sure this happens with probability at most $2t/|K|$ (by sacrificing enough triples in the process), then the error probability of the construction is bounded by $2t/|K|$ for a single run of the test. In order to get negligible error probability we reapeat this phase enough times.

An important fact to notice is that in this construction we need $2t + 1 \leq \mathbb{F}_{p^k}$, since otherwise there are not enough elements to evaluate the polynomials. In order to circumvent this restriction, one can still apply the above construction but replacing $\mathbb{F}_{p^k}$ with an extension $\mathbb{F}_{p^{k'}}$ with the required property.

Asymptotically, we see that as we increase the number $t+1$ of triples checked, we always need to sacrifice t triples, and in addition the number we need to check the multiplication(s) in K. If we assume that we want to hit the desired error probability with just one iteration of the test, we have $2^{-\mathsf{sec}} = 2t/|K|$ from which we get $\log |K| = \mathsf{sec} + \log 2t$. The degree of the extension to K is $\log |K| / \log p^k$, and the number of basic secure multiplications we need is at most the square of this number, which is $(\mathsf{sec} + \log 2t)^2 / (\log p^k)^2$. For each of these, we need error essentially $2^{-\mathsf{sec}}$, so the number of triples we need, say m, satisfies $2^{-\mathsf{sec}} = (1/p^k)^m$, so we get $m = \mathsf{sec}/\log p^k$. This in total grows only poly-logarithmically with t, so we conclude that for a given desired error probability, the number of triples we need to sacrifice to check $t+1$ triples is $O(t + polylog(t))$.

Comparing the two Approaches: A Concrete Example. We here compare the above approaches for checking triples. Suppose $p = 2$ and $k = 8$, so $\mathbb{F}_{p^k} = \mathbb{F}_{2^8}$. Suppose there are also $t + 1 = 128$ triples to check with security level of 2^{-80}.

Using the latter approach, with $K = \mathbb{F}_{2^{16}}$, we need to sacrifice $t = 127$ triples to generate $\langle c_{t+2} \rangle, \ldots, \langle c_{2t+1} \rangle$; moreover we need to perform 4 secure multiplications to check if $A(z)B(z) = C(z)$, since K is a vector space of dimension 2 over $\mathbb{F}_{2^8}$. In order for the multiplications to be secure enough, we need them to be correct up to error probability $(2 \cdot 127)/2^{16} \approx 2^{-8}$ for the entire multiplication $A(z)B(z)$. This will be the case if for each of the 4 small multiplications we use 3 triples for the multiplication, namely one to do the actual multiplication an two to check the first one. This gives a total error of at most $4 \cdot 2^{-16} \leq 2^{-8}$. So since one run of the test leads to an error probability of $\approx 2^{-8}$, we need 10 runs to decrease the error probability to 2^{-80}. Therefore, the total number of triples to sacrifice is $128 + 4 \cdot 3 \cdot 10 = 248$, while with the original approach the number of triples to sacrifice would have been $128 \cdot 10 = 1280$.

A.5 Preprocessing Phase

Proof (Theorem 3).

Recall first that we assume the cryptosystem has an alternative key generation algorithm $\mathsf{KeyGen}^*()$ which is a randomized algorithm that outputs a *meaningless public key* $\widetilde{\mathsf{pk}}$ with the property that an encryption of any message $\mathsf{Enc}_{\widetilde{\mathsf{pk}}}(\mathbf{x})$ is statistically indistinguishable from an encryption of 0. Furthermore, if we set $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$ and $\widetilde{\mathsf{pk}} \leftarrow \mathsf{KeyGen}^*()$, then pk and $\widetilde{\mathsf{pk}}$ are computationally indistinguishable.

We construct a simulator $\mathcal{S}_{\text{PREP}}$ for Π_{PREP}. In a nutshell, the simulator will run a copy of the protocol. Here, it will play the honest players' part while the environment $\mathcal{Z}$ plays for the corrupt players. The simulator also internally runs copies of $\mathcal{F}_{\text{KeyGen}}$ and $\mathcal{F}_{\text{Rand}}$, in order to simulate calls to these functionalities. Note that in the following we say that the simulator executes or performs some part of the protocol as shorthand for the simulator going through that part with $\mathcal{Z}$. During the protocol execution, whenever $\mathcal{Z}$ sends ciphertexts on behalf of corrupt players, the simulator can obtain the plaintexts, since it knows the secret key. These values are then used to generate input to $\mathcal{F}_{\text{PREP}}$. A precise description is provided in Figure 12.

We now need to show that no $\mathcal{Z}$ can distinguish between the simulated and the real process. By contradiction, we assume that there exists $\mathcal{Z}$ that can distinguish these two cases with significant advantage ϵ. The output of $\mathcal{Z}$ is a single bit, thought of a as guess at one of the two cases. Concretely, we assume

$$\begin{aligned}\mathsf{A}(\mathcal{Z}) &:= \Pr\left[\text{“Real”} \leftarrow \mathcal{Z}(\text{Real process})\right] - \Pr\left[\text{“Real”} \leftarrow \mathcal{Z}(\text{Simulated process})\right] \\ &\geq \epsilon.\end{aligned}$$

We will show that such $\mathcal{Z}$ can be used to distinguish between a normally generated public key and a meaningless one with basically the same advantage. This leads to a contradiction, since a key generated by the normal key generator is computationally indistinguishable from a meaningless one.

More in detail, we construct an algorithm B that takes as input a public key pk^* (randomly chosen as either a normal public key or a meaningless one), sets up a copy of $\mathcal{Z}$, goes through the protocol with $\mathcal{Z}$ and uses its output to guess the type of key it got as input. During the process B uniformly chooses a bit (that can be thought as a switch between “Real” and “Simulation”): in

Simulator $\mathcal{S}_{\text{PREP}}$

$\mathsf{SReshare}(e_{\mathbf{m}})$**:** This is a subroutine the simulator will use while executing the main steps of the protocol described below. Any time in Π_{PREP}, when there is a call to $\mathsf{Reshare}(e_{\mathbf{m}})$, the simulator proceeds as the protocol, but it performs the following extra tasks in order to retrieve the quantity $\Delta_{\mathbf{m}}$:

- On step 2 the simulator decrypts $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{f}_1), \ldots, \mathsf{Enc}_{\mathsf{pk}}(\mathbf{f}_n)$ and obtains the values $\mathbf{f}_1, \ldots, \mathbf{f}_n$
- On step 5 the simulator performs step 2 of $\mathcal{F}_{\text{KeyGenDec}}$, and thereby obtains $\mathbf{m}+\mathbf{f}$ decrypting $e_{\mathbf{m}+\mathbf{f}}$, and $(\mathbf{m}+\mathbf{f})'$ from the adversary
- The simulator sets $\Delta_{\mathbf{m}} \leftarrow (\mathbf{m}+\mathbf{f})' - (\mathbf{m}+\mathbf{f})$, that is $\Delta_{\mathbf{m}}$ is the difference between the output chosen by the adversary for the decryption of $e_{\mathbf{m}+\mathbf{f}}$ and the decryption itself.
- The simulator computes and stores $\mathbf{m}_1 \leftarrow (\mathbf{m}+\mathbf{f})' - \mathbf{f}_1$, and $\mathbf{m}_i \leftarrow -\mathbf{f}_i$ for $i \neq 1$.

Initialize:

- The simulator performs the initialization steps of Π_{PREP}. The call to $\mathcal{F}_{\text{KeyGenDec}}$ in step 1 is simulated by running KeyGen to generate the key pair $(\mathsf{pk}, \mathsf{sk})$. The simulator then sends pk to the players and stores sk.
- Steps 2–5 are performed according to the protocol, but the simulator decrypts every broadcast ciphertext and obtains $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$
- Step 6 is performed according to the protocol, but the simulator gets $\Delta_1 \leftarrow \mathsf{SReshare}(e_{\gamma(\alpha\cdot\beta_1)}), \ldots, \Delta_n \leftarrow \mathsf{SReshare}(e_{\alpha\cdot\beta_n})$
- The simulator calls Initialize on $\mathcal{F}_{\text{PREP}}$ with input $\{\alpha_i\}_{i\in A}$ at step 1, $\{\beta_i\}_{i\in A}$ at step 3 and $\Delta_1, \ldots, \Delta_n$ at step 5

Pair:

- The simulator performs step 1 according to the protocol
- Steps 2–3 are performed according to the protocol, but the simulator decrypts every broadcast ciphertext and obtains $\mathbf{r}_1, \ldots, \mathbf{r}_n$
- Step 4 is performed according to the protocol, but the simulator gets $\Delta \leftarrow \mathsf{SReshare}(e_{\mathbf{r}\cdot\alpha})$, $\Delta_1 \leftarrow \mathsf{SReshare}(e_{\mathbf{r}\cdot\beta_1}), \ldots, \Delta_n \leftarrow \mathsf{SReshare}(e_{\mathbf{r}\cdot\beta_n})$
- The simulator calls Pair on $\mathcal{F}_{\text{PREP}}$ with input $\{\mathbf{r}_i\}_{i\in A}$ at step 1, and $\Delta, \Delta_1, \ldots, \Delta_n$ at step 3

Triple:

- The simulator performs step 1 according to the protocol
- Steps 2–3 are performed according to the protocol, but the simulator decrypts every broadcast ciphertext and obtains $\mathbf{a}_1, \ldots, \mathbf{a}_n$, $\mathbf{b}_1, \ldots, \mathbf{b}_n$
- Steps 4–5 are performed according to the protocol, but the simulator gets $\Delta_{\mathbf{a}} \leftarrow \mathsf{SReshare}(e_{\mathbf{a}\cdot\alpha})$, $\Delta_{\mathbf{b}} \leftarrow \mathsf{SReshare}(e_{\mathbf{b}\cdot\alpha})$
- Steps 6–7 are performed according to the protocol, but the simulator gets $\mathbf{c}_1, \ldots \mathbf{c}_n$ and $\delta \leftarrow \mathsf{SReshare}(e_{\mathbf{c}})$
- Step 8 is performed according to the protocol, but the simulator gets $\Delta_{\mathbf{c}} \leftarrow \mathsf{SReshare}(e_{\mathbf{c}\cdot\alpha})$
- The simulator calls Triple on $\mathcal{F}_{\text{PREP}}$ with input $\{\mathbf{a}_i\}_{i\in A}$, $\{\mathbf{b}_i\}_{i\in A}$ at step 1, $\Delta_{\mathbf{a}}, \Delta_{\mathbf{b}}, \delta$ at step 3, $\{\mathbf{c}_i\}_{i\in A}$ in step 5, and $\Delta_{\mathbf{c}}$ at step 7

Fig. 12. The simulator for $\mathcal{F}_{\text{PREP}}$.

case pk^* is correctly computed, if the bit is set to "Real", $\mathcal{Z}$'s view is indistinguishable from a real execution of the protocol, while if the bit is set to "Simulation", $\mathcal{Z}$'s view is indistinguishable from a simulated run. However, in case pk^* is meaningless, both choices of the bit lead to statistically indistinguishable views. Hence, if $\mathcal{Z}$ guesses correctly whether B chose "Real" or "Simulation", B guesses that pk^* was a standard public key; otherwise B guesses that pk^* was meaningless.

For simplicity we describe the algorithm B for the two-party setting, where there is a corrupt party P_1 and an honest party P_2: On input pk^*, where pk^* is a public key (either meaningless or standard), B starts executing the protocol Π_{PREP}, playing for P_2, while $\mathcal{Z}$ plays for P_1. B does exactly what the simulator would do, with some exceptions:

1. It uses the public key it got as input, instead of generating a key pair initially.
2. B cannot decrypt ciphertexts from P_1 since it does not know the secret key (e.g. at step 4 of Initialize, step 2 of Pair, step 2 of Triple, etc.). Instead, B exploits that P_1 and P_2 ran the

protocol Π_{ZKPoPK} with P_1 as prover. That is, P_1 proved that he knows encodings of appropriate size corresponding to the plaintext inside the ciphertexts broadcast in the previous step. This means B can use the knowledge extractor of the protocol Π_{ZKPoPK} followed by decoding to extract the shares from P_1 (e.g. α_i, β_i at step 4 of Initialize, etc). At this point B continues the protocol as if it had decrypted. Note that the knowledge extractor requires rewinding of the prover (which here effectively is $\mathcal{Z}$). B can do this as it runs its own copy of $\mathcal{Z}$ and since it also controls the copy of $\mathcal{F}_{\text{RAND}}$ used in the protocol, it can issue challenges of its choice to $\mathcal{Z}$.

3. When P_2 gives a ZK proof for a set of ciphertexts, B will simulate the proof. This is done by running the honest verifier simulator to get a transcript $(\mathbf{a}, \mathbf{e}, (\mathbf{z}, T))$ and letting the copy of $\mathcal{F}_{\text{RAND}}$ output $\mathbf{e}$ that occurs in the simulate transcript.

In the end B uniformly chooses to generate a real or a simulated view. In the first case, B outputs to $\mathcal{Z}$ exactly those values for P_2 that were used in the execution of the protocol. In the other case, B generates the output for P_2 as $\mathcal{F}_{\text{PREP}}$ would do. That means that P_2's shares $\mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2$ of a triple $\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, \langle \mathbf{c} \rangle$ will be determined by choosing $\mathbf{a}, \mathbf{b}$ at random, setting $\mathbf{c} \leftarrow \mathbf{a} \cdot \mathbf{b}$ and then letting $\mathbf{a}_2 \leftarrow \mathbf{a} - \mathbf{a}_1^{Real}$, $\mathbf{b}_2 \leftarrow \mathbf{b} - \mathbf{b}_1^{Real}$, $\mathbf{c}_2 \leftarrow \mathbf{c} - \mathbf{c}_1^{Real}$.

It can now be seen that if pk^* is a normal key, then the view generated by B corresponds statistically to either a real or a simulated execution: if B chooses the simulation case, the only differences to the actual simulator are 1) the simulator executes the ZK proofs given by P_2 according to the protocol while B simulates them; and 2) the simulator opens the ciphertexts using the secret key to decrypt, while B uses the extractor for Π_{ZKPoPK} and computes the plaintexts from its results. As for 1) the ZK proof is statistical ZK so this leads to a statistically indistinguishable distribution. As for 2), note that for every ciphertext $e_{\mathbf{x}}$ generated by P_1, the extractor for Π_{ZKPoPK} will, except with negligible probability, be able to find an encoding $\mathbf{x}$ (resp. randomness r) smaller than B_{plain} (resp. B_{rand}), with $e_{\mathbf{x}} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}, \mathbf{r})$. This follows from soundness of Π_{ZKPoPK} and admissibility of the cryptosystem. Then, by correctness of the cryptosystem, computing the plaintexts as B does, will indeed give the same result as decrypting, except with negligible probability. If B chooses the real case, a similar argument shows that we get a view statistically indistinguishable from a real run of the protocol. Hence if pk^* is a normal key, $\mathcal{Z}$ can guess B's choice of "Real" or "Simulation" with advantage essentially ϵ.

On the other hand if pk^* is a meaningless key, the encryptions contain statistically no information about the values inside. Moreover, all messages sent in the zero-knowledge protocols where P_2 acts as prover, do not depend on the specific values that P_2 has, since the proofs are simulated. We conclude that essentially no information on any value held by P_2 is revealed. This is the case also for step 5 of $\mathsf{Reshare}(e_{\mathbf{m}})$: $\mathbf{m}+\mathbf{f}$ is retrieved, but no information on $\mathbf{m}$ is revealed, since $\mathbf{f}$ is uniform.

The view $\mathcal{Z}$ sees consists of the view of the corrupt player(s) and the output of the honest player(s). We just argued that the view of the corrupt player is essentially independent of the internal values B uses for P_2, and hence also independent of whether B chooses the real or the simulated case. Therefore, the output generated for the honest player(s) seen by $\mathcal{Z}$ is in both cases a set of (essentially) uniformly and independently chosen shares and MAC keys. As a result, if we use a meaningless key, a real execution and a simulated execution are statistically indistinguishable, and the guess of $\mathcal{Z}$ will equal B's random choice of "Real" or "Simulation" with probability essentially $1/2$.

An easy calculation now shows that the advantage of B is

$$\begin{aligned}\mathsf{A}(B) &:= \Pr\left[\text{“Standard Key”} \leftarrow B(\mathsf{pk})\right] - \Pr\left[\text{“Standard Key”} \leftarrow B(\widetilde{\mathsf{pk}})\right] \\ &\geq \mathsf{A}(\mathcal{Z})/2 - \delta \\ &= \epsilon/2 - \delta,\end{aligned}$$

for some negligible δ that accounts for the differences between the involved distributions. However, if ϵ is non-negligible, then $\epsilon/2 - \delta$ is also non-negligible, which contradicts the assumption on that meaningless keys are statistically indistinguishable from standard ones. □

A.6 Distributed Decryption

Proof (Theorem 4). The requirement $B + 2^{\mathsf{sec}} \cdot B < q/2$ implies that $\mathbf{t}' = \mathbf{t} \bmod p$, since $\|\mathbf{r}_i\|_\infty < 2^{\mathsf{sec}} \cdot B/(n \cdot p)$ for $i = 1, \ldots, n$. Therefore the protocol allows players to retrieve the correct message if all the players are honest.

We now build a simulator $\mathcal{S}_{\text{DDEC}}$ to work on top of $\mathcal{F}_{\text{KEYGENDEC}}$, such that the adversary cannot distinguish whether it is playing with the decryption protocol and $\mathcal{F}_{\text{KEYGEN}}$ or the simulator and $\mathcal{F}_{\text{KEYGENDEC}}$. We let A denote the set of players controlled by the adversary.

Simulator $\mathcal{S}_{\text{DDEC}}$

Key Generation: This stage is needed to distribute shares of a secret key.
- Upon “start”, the simulator sends “start” to $\mathcal{F}_{\text{KEYGENDEC}}$ and obtains pk. Moreover, the simulator obtains $(\mathsf{sk}_i)_{i \in A}$ from the adversary.
- The simulator (internally) sets random $(\mathsf{sk}_i)_{i \notin A}$ such that $(\mathsf{sk}_i)_{i=1,\ldots,n}$ is a full vector of shares of 0.
- The simulator sends pk to A.

Public Decryption: This stage simulates a public decryption.
- Upon “decrypt c, B”, the simulator sends “decrypt c” to $\mathcal{F}_{\text{KEYGENDEC}}$ and obtains $m = \mathsf{Dec}_{\mathsf{sk}}(c)$.
- It then computes the value $\mathbf{v}_i$ for all players except for an honest player P_j.
- It then samples $\mathbf{r}_j$ uniformly with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$ and computes

$$\widetilde{\mathbf{t}}_j \leftarrow -\sum_{i \neq j} \mathbf{v}_i + p \cdot \mathbf{r}_j + \mathsf{encode}(m).$$

- For each other honest player P_i, it computes $\mathbf{t}_i$ honestly (using c, sk_i).
- The simulator broadcasts the values $(\mathbf{t}_i)_{i \notin A, i \neq j}, \widetilde{\mathbf{t}}_j$ and obtains $(\mathbf{t}_i^*)_{i \in A}$ from the adversary.
- It then sends $m' \leftarrow \mathsf{decode}\left(\left(\widetilde{\mathbf{t}}_j + \sum_{i \in A} \mathbf{t}_i^* + \sum_{i \notin A, i \neq j} \mathbf{t}_i\right) \bmod p\right)$ to $\mathcal{F}_{\text{KEYGENDEC}}$ so that the ideal functionality sends “Result m'” to all the players.

Private Decryption: This stage simulates a private decryption.
- Upon “decrypt c, B to P_j”, the simulator sends “decrypt c to P_j” to $\mathcal{F}_{\text{KEYGENDEC}}$.
- If P_j is corrupt, the simulator obtains $c, m = \mathsf{Dec}_{\mathsf{sk}}(c)$ from $\mathcal{F}_{\text{KEYGENDEC}}$ and acts as in the simulated public decryption.
- If P_j is honest, the simulator receives c from $\mathcal{F}_{\text{KEYGENDEC}}$, $\mathbf{t}_i^*$ from each corrupt player P_i and $\mathbf{t}_i$ from each honest player.
  - The simulator samples $\mathbf{r}_j$ uniformly with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$.
  - It evaluates $\widetilde{\mathbf{t}}_j \leftarrow -\sum_{i \neq j} \mathbf{v}_i + p \cdot \mathbf{r}_j$.
  - It computes $\varepsilon \leftarrow \left(\widetilde{\mathbf{t}}_j + \sum_{i \in A} \mathbf{t}_i^* + \sum_{i \notin A, i \neq j} \mathbf{t}_i\right) \bmod p$
  - Finally it sends $\delta \leftarrow \mathsf{decode}(\varepsilon)$ to $\mathcal{F}_{\text{KEYGENDEC}}$ in order to get $\mathsf{Dec}_{\mathsf{sk}}(c) + \delta$ to P_j.

Fig. 13. The simulator for Π_{DDEC}.

In a simulated decryption the adversary receives pk and $(\mathbf{t}_i)_{i\notin A, i\neq j}, \widetilde{\mathbf{t}}_j$ from $\mathcal{S}_{\mathrm{DDEC}}$. The distribution of pk is the same as in a real conversation, since it was sampled using the same algorithm as in a real conversation. The distribution of simulated $\mathbf{t}_i$, $i \neq j$ is statistically close to the real one, since $\mathbf{t}_i$ was computed correctly using shares of a possible secret key. We can therefore focus on the case where all the players but one are dishonest. We first analyse the simulation of public decryption, introducing a hybrid machine, and prove its output is statistically indistinguishable from P_j's output (in the real protocol) and perfectly indistinguishable from P_j's simulated output.

Hybrid: On input $(\mathsf{sk}_i)_{i=1,\dots,n}, c$, reconstruct sk, compute $\mathsf{Dec}_{\mathsf{sk}}(c)$, sample $\mathbf{r}_j$ uniformly with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$ and output $\widetilde{\mathbf{t}}_j \leftarrow -\sum_{i\neq j} \mathbf{v}_i + p \cdot \mathbf{r}_j + \mathsf{encode}(m)$.

Notice that $\widetilde{\mathbf{t}}_j = \mathbf{v}_j - \mathbf{t} + \mathsf{encode}(m) + p \cdot \mathbf{r}_j$. Now, for a distribution X, define $\varphi(X) := p \cdot X + \mathbf{v}_j$. Notice that $\mathbf{t}_j = \varphi(U)$, where U denotes the uniform distribution over vectors of integral entries bounded with infinity norm $2^{\mathsf{sec}} \cdot B$; moreover, since $\mathbf{t} - \mathsf{encode}(m)$ is a multiple of p, one can write $\widetilde{\mathbf{t}}_j = \varphi(U + (\mathsf{encode}(m) - \mathbf{t})/p)$. Since $\|(\mathsf{encode}(m) - \mathbf{t})/p\|_\infty \leq (B+1)/p$ and U is uniform in an exponentially larger range, then the distribution $U + (\mathsf{encode}(m) - \mathbf{t})/p$ is statistically close to U. Therefore $\widetilde{\mathbf{t}}_j$ is statistically close to $\mathbf{t}_j$.

What is left to prove is that the simulation of private decryption to an honest player P_j is statistically indistinguishable from the real protocol. In the real protocol P_j computes $\mathbf{t}_j$ and

$$m' \leftarrow \mathsf{decode}\left(\sum_{i\in A} \mathbf{t}_i^* + \sum_{i\notin A} \mathbf{t}_i\right).$$

In that case the error $m' - m$ introduced by the adversary depends only on the value

$$\varepsilon' := \left(\sum_{i\in A} (\mathbf{t}_i^* - \mathbf{t}_i)\right) \bmod p$$

computed using the actual secret key. In the simulation the error introduced by the adversary is

$$\varepsilon = \left(\widetilde{\mathbf{t}}_j + \sum_{i\in A} \mathbf{t}_i^* + \sum_{i\notin A, i\neq j} \mathbf{t}_i\right) \bmod p = \left(\sum_{i\in A} (\mathbf{t}_i^* - \mathbf{t}_i)\right) \bmod p,$$

computed using secret shares of 0. Since the secret sharing scheme has privacy threshold n and the sums involve at most $n-1$ shares, the quantities ε and ε' are statistically indistinguishable. □

B A lower Bound for the Preprocessing

In this section, we show that any preprocessing matching the properties we have, must output the same amount of data as we do, up to a constant factor. We use the following theorem for 2-party computation from [29]. It talks about a setting where the parties A, B have access to a functionality that gives a random variable U to A and V to B with some guaranteed joint distribution P_{UV} of U, V. Given this, the parties compute securely a function $f : \mathcal{X} \times \mathcal{Y} \mapsto \mathcal{Z}$, where A holds $x \in \mathcal{X}$, and B holds $y \in \mathcal{Y}$. This function should have the property that there exists inputs y_1, y_2 such that for all $x \neq x'$, $f(x, y_1) \neq f(x', y_1)$; and for all x, x', $f(x, y_2) = f(x', y_2)$. In other words, for some inputs B learns all of A's input, but other inputs B learns nothing new.

Theorem 6. *Let $f : \mathcal{X} \times \mathcal{Y} \mapsto \mathcal{Z}$ be a function with inputs y_1, y_2 as above. If there exists a protocol that computes f securely with access to P_{UV} and with error probability ϵ in the semi-honest model, then*

$$H(V) \geq I(U;V) \geq \log|\mathcal{X}| - 7(\epsilon \log|\mathcal{X}| + h(\epsilon))$$

We will also need the following technical lemma

Lemma 1. *Let R be a random variable defined over the natural numbers. Then there exists a constant C such that $E(R) \geq H(R) - 1 - C$.*

Proof (Lemma 1). Let

$$I := \left\{ i \mid i \geq \log\left(\frac{1}{Pr[R=i]}\right) \right\}.$$

Under such a definition, one can write $H(R)$ as

$$H(R) = \sum_{i \in I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right) + \sum_{i \notin I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right)$$

By the construction of I, one can bound the first summand as follows

$$\begin{aligned} \sum_{i \in I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right) &\leq \sum_{i \in I} Pr[R=i] \cdot i \\ &\leq \sum_{i} Pr[R=i] \cdot i \\ &= E(R). \end{aligned}$$

For the second summand one needs to work a bit more. Let $q(i) := \log(1/Pr[R=i])$. Then

$$\sum_{i \notin I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right) = \sum_{i \notin I} 2^{-q(i)} \cdot q(i).$$

We now claim that

$$2^{-q(i)} \cdot q(i) \leq 2^i \cdot i, \text{ for all } 0 \neq i \notin I.$$

This happens if and only if

$$2^{-q(i)} \cdot 2^{\log(q(i))} \leq 2^{-i} \cdot 2^{\log(i)}.$$

Taking the logarithm of such relation one gets $-q(i) + \log(q(i)) \leq -i + \log(i)$, which is equivalent to $q(i) - \log(q(i)) \geq i - \log(i)$.
Since $q(i) = \log(1/Pr[R=i]) \geq i$ for all $i \notin I$, and $i \geq 1$, the latter relation is always satisfied. Therefore, one can bound the second summand by $C + \sum_{i \geq 1} 2^{-i} \cdot i$, where $C = 2^{-q(0)} \cdot q(0)$.
Moreover $\sum_{i \geq 1} 2^{-i} \cdot i$ converges to 1, so the second summand can be bound by $1 + C$.

Finally, one can reassemble all the reasoning into one and get

$$\sum_{i \in I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right) + \sum_{i \notin I} Pr[R=i] \cdot \log\left(\frac{1}{Pr[R=i]}\right) \leq E(R) + C + 1.$$

The last inequality implies that $H(R) \leq E(R) + 1 + C$ □

With this result, we can prove the lower bound claimed earlier:

Proof (Theorem 2). Suppose we have an on-line protocol π that satisfies the assumptions in the theorem. Consider any player P_i and suppose we want to compute the function

$$f_T((\boldsymbol{x}, \boldsymbol{x}'), y) = y\boldsymbol{x} + (1-y)\boldsymbol{x}'.$$

Here $y \in \mathbb{F}_{p^k}$ and $\boldsymbol{x}, \boldsymbol{x}'$ are vectors over $\mathbb{F}_{p^k}$ of length T. P_i will have input y and each P_j, $j \neq i$ will have as input substrings $\boldsymbol{x}_j, \boldsymbol{x}'_j$ such that the concatenation of all $\boldsymbol{x}_j$ ($\boldsymbol{x}'_j$) is $\boldsymbol{x}$ ($\boldsymbol{x}'$). Finally, only P_i learns the output $f_T((\boldsymbol{x}, \boldsymbol{x}'), y)$.

Clearly, f_T can be computed using a circuit of size $O(T)$, and this will be the circuit promised in the theorem. Note that our assumed protocol π can handle circuits of size S and can therefore compute f_T securely where T is $\Theta(S)$.

We can now transform π to a two-party protocol π' for parties A and B. A has input $\boldsymbol{x}, \boldsymbol{x}'$, B has input y and B is supposed to learn $f_T((\boldsymbol{x}, \boldsymbol{x}'), y)$. Now, π' simply consists of running π where B emulates P_i and A emulates all other players. We give to B whatever P_i gets from the preprocessing and A gets whatever the other players receive, so this defines the random variables U and V. Since π is secure if P_i is corrupt and also if all other players are corrupt, this trivially means that π' is an actively secure two-party protocol for computing f_T.

This implies that π' also computes f_T with passive security. As noted in [29], this is actually not necessarily the case for all functions. The problem is that if the adversary is passive, then active security does guarantees that there is a simulator for this case, but such a simulator is allowed to change the inputs of corrupted parties. A simulator for the passive case is not allowed to do this. However, [29] observe that for some functions, an active simulator cannot get away with changing the inputs, as this would make it impossible to simulate correctly. They show this is the case for Oblivious Transfer which is essentially what f_T is after we go to the 2-party case. We may therefore assume π' is also passively secure.

Finally, we define $f'_T(\boldsymbol{x}, y) = f_T((\boldsymbol{x}, \boldsymbol{0}), y) = y\boldsymbol{x}$. Obviously π' can be used to compute f'_T securely, A just sets her second input to be $\boldsymbol{0}$. Moreover f'_T satisfies the conditions in Theorem 6. So we get that $H(V) \geq \log|\mathcal{X}| - 7(\epsilon \log|\mathcal{X}| + h(\epsilon))$. If we adopt the standard convention that the security parameter grows linearly with the input size $\log|\mathcal{X}|$ then because ϵ is negligible in the security parameter, we have that the "error term" $7(\epsilon \log|\mathcal{X}| + h(\epsilon))$ is $o(\log|\mathcal{X}|)$.

So we get that $H(V)$ is $\Omega(\log|\mathcal{X}|) = \Omega(T \log p^k) = \Omega(S \log p^k)$, since T is $\Theta(S)$. Recalling that $H(V)$ is actually the entropy of the variable P_i received in the original protocol π, we get the first conclusion of the Theorem.

For the second conclusion about the computational work done, it is tempting to simply claim that B has to at least read the information he is given and so $H(V)$ is a lower bound on the expected number of bit operations. But this is not enough. It is conceivable that in every particular execution, B might only have to read a small part of the information.

It turns out that this does not happen, however, which can be argued as follows: let $B(V)$ be the random variable representing the bits of V that B actually reads. By inspection of the proof of Theorem 6, one sees that if we replace everywhere V by $B(V)$ the same proof still applies. So in fact, we have $H(B(V)) \geq \log|\mathcal{X}| - 7(\epsilon \log|\mathcal{X}| + h(\epsilon))$. Now let R be the random variable representing the number of bits B reads from V.

If we condition on R, then the entropy of $B(V)$ cannot drop by more than $H(R)$, so we have

$$H(B(V)|R) \geq H(B(V)) - H(R) \geq \log|\mathcal{X}| - 7(\epsilon \log|\mathcal{X}| + h(\epsilon)) - H(R).$$

Moreover, we also have

$$H(B(V)|R) = \sum_r Pr(R=r)H(B(V)|R=r) \leq \sum_r Pr(R=r)r = E(R)$$

Putting these two inequalities together, we obtain that

$$E(R) + H(R) \geq \log|\mathcal{X}| - 7(\epsilon \log|\mathcal{X}| + h(\epsilon)).$$

Now, either $E(R) \geq (\log|\mathcal{X}| - 7(\epsilon\log|\mathcal{X}| + h(\epsilon)))/2$, or $H(R) \geq (\log|\mathcal{X}| - 7(\epsilon\log|\mathcal{X}| + h(\epsilon)))/2$. In the latter case we have from Lemma 1 that $E(R)$ is much larger than $H(R)$, so we can certainly conclude that $E(R) \geq (\log|\mathcal{X}| - 7(\epsilon\log|\mathcal{X}| + h(\epsilon)))/2$ in any case. As above, the error term depending on ϵ becomes negligible for increasing security parameter, so we get that $E(R)$ is $\Omega(S\log p^k)$ as desired. □

C Canonical Embeddings of Cyclotomic Fields

Our concrete instantiation will use some basic results of Cyclotomic fields which we now recap on; these results are needed for the main result of this Appendix which is a proof of a "folklore" result about the relationship between norms in the canonical and polynomial embeddings of a cyclotomic field. This result is used repeatedly in our main construction to produce estimates on the size of parameters needed.

C.1 Cyclotomic Fields

We first recap on some basic facts about numbers fields, and their canonical embeddings. Focusing particularly on the case of cyclotomic fields.

Number Fields An algebraic number (resp. algebraic integer) $\theta \in \mathbb{C}$ is the root of a polynomial (resp. monic polynomial) with coefficients in $\mathbb{Q}$ (resp. $\mathbb{Z}$). The minimal polynomial of θ is the unique monic irreducible $f(x) \in \mathbb{Q}[X]$ which has θ as a root.

A number field $K = \mathbb{Q}(\theta)$ is the field obtained by adjoining powers of an algebraic number θ to $\mathbb{Q}$. If θ has minimal polynomial $f(x)$ of degree N, then K can be considered as a vector space over $\mathbb{Q}$, of dimension N, with basis $\{1, \theta, \ldots, \theta^{N-1}\}$. Note that this "coefficient embedding" is relative to the defining polynomial $f(x)$ Equivalently we have $K \cong \mathbb{Q}[X]/f(X)$, i.e. the field of rational polynomials with degree less than N, modulo the polynomial $f(X)$. Without loss of generality we can assume K, from now on, is defined by a monic irreducible integral polynomial of degree N. The ring of integers $\mathcal{O}_K$ of K is defined to be the subring of K consisting of all elements whose minimal polynomial has integer coefficients.

Canonical Embedding There are N field morphisms $\sigma_i : K \longrightarrow \mathbb{C}$ which fix every element of $\mathbb{Q}$. Such a morphism is called a complex *embedding* and it takes θ to each distinct complex root of $f(X)$. The number field K is said to have signature (s_1, s_2) if the defining polynomial has s_1 real roots and s_2 complex conjugate pairs of roots; clearly $N = s_1 + 2 \cdot s_2$. The roots are numbered in the standard way so that $\sigma_i(\theta) \in \mathbb{R}$ for $1 \leq i \leq s_1$ and $\sigma_{i+s_1+s_2}(\theta) = \overline{\sigma_{i+s_1}(\theta)}$ for $1 \leq i \leq s_2$. We define $\sigma = (\sigma_1, \ldots, \sigma_N)$, which defines the *canonical embedding* of K into $\mathbb{R}^{s_1} \times \mathbb{C}^{2\cdot s_2}$, where the field operations in K are mapped into componentwise addition and multiplication in $\mathbb{R}^{s_1} \times \mathbb{C}^{2\cdot s_2}$. To ease notation we will often write $\alpha^{(i)} = \sigma_i(\alpha)$, for $\alpha \in K$. We will let $\|\alpha\|_p$ for $p \in [1, \ldots, \infty]$ denote the p-norm of α in the coefficient embedding (i.e. the p-norm of the vector of coefficients) and let $\|\sigma(\alpha)\|_p$ denote norms in the canonical embedding.

Cyclotomic Fields We will mainly be concerned with cyclotomic number fields. The mth cyclotomic polynomial is given by $\Phi_m(X)$, this is an irreducible polynomial of degree $N = \phi(m)$. The number field defined by $\Phi_m(X)$ is said to be a cyclotomic number field, and is defined by $K = \mathbb{Q}(\zeta_m)$, where ζ_m is an mth root of unity, i.e. a root of $\Phi_m(X)$. The ring of integers of K is equal to $\mathbb{Z}[\zeta_m]$. The number field K is Galois, and hence (importantly for us) the polynomial splits modulo p (for any prime p not dividing m) into a produce of distinct irreducible polynomials all of the same degree.

The key fact is that if $\Phi_m(X)$ has degree d factors modulo the prime p then m divides $p^d - 1$. To see this notice that if $\Phi_m(X)$ factors into N/d factors each of degree d then the finite field $\mathbb{F}_{p^d}$ must contain the mth roots of unity and so m divides $p^d - 1$. In the other direction, if d is the smallest integer such that m divides $p^d - 1$ then $\Phi_m(X)$ will have a degree d factor since the decomposition group of the prime p in the Galois group will have order d.

C.2 Relating Norms Between Canonical and Polynomial Embeddings

There is a distinct difference between the canonical and polynomial embeddings of a number field. In particular notice the following expansions upon multiplication, for $x, y \in \mathcal{O}_K$,

$$\|x \cdot y\|_\infty \leq \delta_\infty \cdot \|x\|_\infty \cdot \|y\|_\infty.$$
$$\|\sigma(x \cdot y)\|_p \leq \|\sigma(x)\|_\infty \cdot \|\sigma(y)\|_p.$$

where

$$\delta_\infty = \sup \left\{ \frac{\|a(X) \cdot b(X) \pmod{f(X)}\|_\infty}{\|a(X)\|_\infty \cdot \|b(X)\|_\infty} : a, b \in \mathbb{Z}[X], \deg(a), \deg(b) < N \right\}.$$

In this section we show that one can more tightly control the expansion factor of elements in the polynomial representation; as long as they are drawn randomly with a discrete Gaussian distribution. In particular we prove the following theorem; this result is well known to people working in ideal lattice theory, but proofs have not yet appeared in any paper.

Theorem 7. *Let K denote a cyclotomic number field then there is a constant C_m, depending only on m, such that for all $\alpha \in \mathcal{O}_K$ we have*

- $\|\sigma(\alpha)\|_\infty \leq \|\alpha\|_1$.
- $\|\alpha\|_\infty \leq C_m \cdot \|\sigma(\alpha)\|_\infty$.

We recall some facts about various matrices associated with roots of unity, see [27] and the full version of [22]. First some notation; for any integer $m \geq 2$: We set $\zeta_m = \exp(2 \cdot \pi \cdot \sqrt{-1}/m)$ to be a root of unity for an integer m. As usual we let $N = \phi(m)$ and we define $\mathbb{Z}_m^* = \{a_{m,i} : 0 \leq i < N\}$ to be a complete set of representatives for $\mathbb{Z}_m^*$ with $1 \leq a_{m,i} < m$. We let $A \otimes B$, for matrices A and B, denote the Kronecker product. We let I_t denote the $t \times t$ identity matrix. All $a \times b$ matrices M in this section will have elements $m_{i,j}$ indexed by $0 \leq i < a$ and $0 \leq j < b$; i.e. we index from zero; this is to make some of the expressions easier to write down. The infinity norm for a matrix $M = (m_{i,j})$ is defined by

$$\|M\|_\infty := \max \left\{ \sum_{j=0}^{N-1} |m_{i,j}| \right\}_{i=0}^{N-1}.$$

We define the $N \times N$ CRT matrix as follows:

$$\mathsf{CRT}_m := \left(\zeta_m^{a_{m,i} \cdot j}\right)_{0 \leq i,j < N}.$$

Then we define the constant C_m in the above theorem as $C_m = \|\mathsf{CRT}_m^{-1}\|_\infty$. From which the proof now immediately follows:

Proof (Theorem 7). For a cyclotomic field the canonical embedding is given by the map $\sigma(\alpha) = \mathsf{CRT}_m \cdot \alpha$, where α is the vector of the coefficient embedding of α, i.e. α considered as a polynomial in θ a root of $F(X) = \Phi_m(X)$ and CRT_m is the matrix, defined earlier, i.e. it is equal to

$$\mathsf{CRT}_m = \begin{pmatrix} 1 & \theta^{(1)} & \dots & \theta^{(1)N-1} \\ \vdots & \vdots & & \vdots \\ 1 & \theta^{(N)} & \dots & \theta^{(N)N-1} \end{pmatrix}.$$

For the first part of the theorem we note that, on writing $\alpha = \sum_{i=0}^{N-1} x_i \cdot \theta^i$, we have

$$\left|\alpha^{(i)}\right| = \left|\sum_{j=0}^{N-1} x_j \cdot \theta^{(i)j}\right| \leq \sum_{j=0}^{N-1} |x_j| \cdot |\theta^{(i)j}| = \sum_{j=0}^{N-1} |x_j| = \|\mathbf{x}\|_1 = \|\alpha\|_1.$$

For the second part we note that for all $\beta \in \mathcal{O}_K$,

$$\|\beta\|_\infty = \|\mathsf{CRT}_m^{-1} \cdot \sigma(\beta)\|_\infty \leq \|\mathsf{CRT}_m^{-1}\|_\infty \cdot \|\sigma(\beta)\|_\infty$$

from which the result follows. □

The key question then is how large can C_m become. So we now turn to this problem; giving a partial answer.

The $m \times m$ DFT matrix is defined by:

$$\mathsf{DFT}_m := \left(\zeta_m^{i \cdot j}\right)_{0 \leq i,j < m}.$$

Let m' be a divisor of m then for $i \in \{0, \dots, m-1\}$ we write $i_0 = i \bmod m'$ and $i_1 = (i - i_0)/m'$. We then define the $m \times m$ "twiddle matrix" to be the diagonal matrix defined by

$$T_{m,m'} := \mathsf{Diag}\left\{\zeta_m^{i_0 \cdot i_1}\right\}_{i=0,\dots,m-1}.$$

Finally we define $L_{m'}^m$ to be the permutation matrix which fixes the row with index $m-1$, but sends all other rows i, for $0 \leq i < m-1$, to row $i \cdot m' \bmod m - 1$. Following [27] we use these matrices to decompose the matrix D_m into D'_m and D_k, where $m = m' \cdot k$, via the following identity

$$\mathsf{DFT}_m = L_{m'}^m \cdot (I_k \otimes \mathsf{DFT}_{m'}) \cdot T_{m,m'} \cdot (\mathsf{DFT}_k \otimes I_{m'}), \tag{3}$$

This is nothing but the general Cooley-Tukey decomposition of the DFT for composite m. Consider the Vandermonde matrix

$$V(x_1, \dots, x_m) := \begin{pmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{m-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{m-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{m-1} \end{pmatrix}.$$

It is clear that $\mathsf{DFT}_m = V(1, \zeta_m, \zeta_m^2, \dots, \zeta_m^{m-1})$.

Lemma 2. *We have, for any m,*

$$\mathsf{DFT}_m^{-1} = \frac{1}{m} \cdot V(1, \zeta_m^{-1}, \zeta_m^{-2}, \ldots, \zeta_m^{1-m})$$

Proof. Let $\delta_{i,j}$ be defined so that $\delta_{i,j} = 0$ if $i \neq j$ and equal to one otherwise. We have

$$\begin{aligned}\left(V(1, \zeta_m, \zeta_m^2, \ldots, \zeta_m^{m-1}) \cdot V(1, \zeta_m^{-1}, \zeta_m^{-2}, \ldots, \zeta_m^{1-m})\right)_{i,j} &= \sum_{0 \leq k < m} \zeta_n^{i \cdot k} \cdot \zeta_m^{-k \cdot j} \\ &= \sum_{0 \leq k < m} \zeta_m^{k \cdot (i-j)} = m \cdot \delta_{i,j}.\end{aligned}$$

□

This leads to the following lemma which gives shows that the infinity norm of the inverse of the DFT matrix is always equal to one.

Lemma 3. *For any m we have $\|\mathsf{DFT}_m^{-1}\|_\infty = 1$.*

Proof. If ζ_m is an m-th root of unity, it is clear that $\|V(1, \zeta_m, \zeta_m^2, \ldots, \zeta_m^{m-1})\|_\infty = m$. In addition we have $\psi_m = 1/\zeta_m$ is also an m-th root of unity, thus

$$\|\mathsf{DFT}_m^{-1}\|_\infty = \frac{1}{m} \cdot \|V(1, \psi_m, \psi_m^2, \ldots, \psi_m^{m-1})\|_\infty = \frac{m}{m} = 1.$$

□

Let $m = p_1^{e_1} \cdots p_k^{e_s}$ we define $r = p_1 \cdots p_s$, $m_1 = m/r$; hence $N = \phi(m) = \phi(r) \cdot m_1$. In [22] the authors specialise the decomposition (3) (by selecting appropriate rows and columns) in the case $m' = m_1$ and $k = r$, to show that to show that, upto a permutation of the rows, the matrix CRT_m is equal to

$$\left(I_{\phi(r)} \otimes \mathsf{DFT}_{m_1}\right) \cdot T^*_{m,m_1} \cdot (\mathsf{CRT}_r \otimes I_{m_1})$$

where ${T_{m,m_1}}^*$ is another diagonal matrix consisting of roots of unity. We then have that

Lemma 4. *For an integer $m \geq 2$ such that $m = p_1^{e_1} \cdots p_k^{e_k}$ we write $r = p_1 \cdots p_k$, we then have $C_m \leq C_r$.*

Proof. As above we write $m_1 = m/r$. First note that $\|A \otimes I_t\|_\infty = \|I_s \otimes A\|_\infty = \|A\|_\infty$ for any matrix A and any integers s and t. Then also note that since CRT_m is given, upto a permutation of the rows, by the above decomposition, we have that CRT_m^{-1} is given up to a permutation of the rows by the decomposition

$$\left(\mathsf{CRT}_r^{-1} \otimes I_{m_1}\right) \cdot T^{-1} \cdot \left(I_{\phi(r)} \otimes \mathsf{DFT}_{m_1}^{-1}\right).$$

So we have

$$\begin{aligned}\|\mathsf{CRT}_m^{-1}\|_\infty &= \| \left(\mathsf{CRT}_r^{-1} \otimes I_{m_1}\right) \cdot T^{-1} \cdot \left(I_{\phi(r)} \otimes \mathsf{DFT}_{m_1}^{-1}\right) \|_\infty, \\ &\leq \|\mathsf{CRT}_r^{-1} \otimes I_{m_1}\|_\infty \cdot \|T^{-1}\|_\infty \cdot \|I_{\phi(r)} \otimes \mathsf{DFT}_{m_1}^{-1}\|_\infty, \\ &= \|\mathsf{CRT}_r^{-1}\|_\infty \cdot \|T^{-1}\|_\infty \cdot \|\mathsf{DFT}_{m_1}^{-1}\|_\infty = \|\mathsf{CRT}_r^{-1}\|_\infty.\end{aligned}$$

□

This result means that we can bound C_m for infinite families of values of m, by simply deducing a bound on C_r, where r is the product of all primes dividing m. For example notice that $\mathsf{CRT}_r = (1)$ and hence $C_{2^e} = C_2 = 1$ for all values of e. Indeed it is relatively straight forward to determine the exact value of C_p for a prime p:

Lemma 5. *If p is a prime then*

$$C_p = \frac{2 \cdot \sin(\pi/p)}{p \cdot (\cos(\pi/p) - 1)}.$$

Proof. [7] First note that it is a standard fact from algebra (by consider inverses of Vandermonde matrices for example) that the entries of a row of the matrix CRT_p^{-1} are given by the coefficients of the polynomial

$$\frac{\Phi_p(X)}{\Phi_p'(\zeta_p) \cdot (X - \zeta_p)}, \tag{4}$$

where each row uses a different root of unity ζ_p. We then note that

$$\begin{aligned}
\Phi_p'(\zeta_p) &= (\zeta_p - \zeta_p^2) \cdot (\zeta_p - \zeta_p^3) \cdots (\zeta_p - \zeta_p^{p-1}) \\
&= \zeta_p^{-2} \cdot (1 - \zeta_p) \cdot (1 - \zeta_p^2) \cdots (1 - \zeta_p^{p-2}) \cdot \frac{(1 - \zeta_p^{p-1})}{(1 - 1/\zeta_p)} \\
&= \frac{\zeta_p^{-2} p}{1 - 1/\zeta_p} = \frac{p}{\zeta_p^2 - \zeta_p}.
\end{aligned}$$

Thus the coefficients of the polynomial in (4) are given by $\zeta_p \cdot (\zeta_p^r - 1)/p$ for $r = 1, \ldots, p-1$. Where each row of our matrix is given by a different pth root ζ_p.

Thus to determine the infinity norm of CRT_p^{-1} we simply need to sum the absolute values of these coefficients, for the first row, since all other rows will be equal:

$$\begin{aligned}
C_p &= \sum_{r=1}^{p-1} |\zeta_p(\zeta_p^r - 1)/p| = \frac{1}{p} \sum_{r=1}^{p-1} \sqrt{2 - 2 \cdot \cos(2r\pi/p)} \\
&= \frac{1}{p} \sum_{r=1}^{p-1} 2 \cdot \sin(r\pi/p) = \frac{2 \cdot \sin(\pi/p)}{p \cdot (\cos(\pi/p) - 1)}
\end{aligned}$$

□

In practice this result means that $C_p \approx 4/\pi \approx 1.2732$ for all $p \geq 11$.

If m is odd then we see that, subject to a permutation of the rows, the matrix CRT_{2m} and CRT_m are identical up to a multiple of -1 for every second column. Thus we have

$$C_{2m} = C_m \text{ for odd values of } m.$$

We find that $C_r \leq 8.6$ for squarefree $r \leq 400$, which provides a relatively small upper bound on C_m for an infinite family of cyclotomic fields K. It appears that the size of C_m depends crucially on the number of prime factors of m. Thus it is an interesting open question to provide a tight upper bound on C_m. Indeed the growth in C_m seems to be closely related to the growth in the coefficients of the polynomial $\Phi_m(X)$, which also depends on the number of prime factors of m.

[7] This proof was provided to us by Robin Chapman .

C.3 Application of the above bounds

An immediate consequence of Theorem 7 is to provide an upper bound on the value δ_∞ for cyclotomic number fields. Let $\alpha \in \mathcal{O}_K$ then we have, by the standard inequalities between norms, that $\|\alpha\|_1 \leq N \cdot \|\alpha\|_\infty$. Thus we have, for $\alpha, \beta \in \mathcal{O}_K$,

$$\begin{aligned}\|\alpha \cdot \beta\|_\infty &\leq C_m \cdot \|\sigma(\alpha \cdot \beta)\|_\infty \leq C_m \cdot \|\sigma(\alpha)\|_\infty \cdot \|\sigma(\beta)\|_\infty \\ &\leq C_m \cdot \|\alpha\|_1 \cdot \|\beta\|_1 \\ &\leq C_m \cdot N^2 \cdot \|\alpha\|_\infty \cdot \|\beta\|_\infty,\end{aligned}$$

i.e. $\delta_\infty \leq C_m \cdot N^2$. When m is a power of two, since $C_m = 1$ we find the bound $\delta_\infty \leq \phi(m)^2$; however in this case it is known that $\delta_\infty = \phi(m)$, thus the above bound is not tight.

A more interesting application, for our purposes, is to bound the infinity norm in the polynomial embedding of the product of two elements which have been selected with a discrete Gaussian. To demonstrate this result we will first need to introduce the following standard tailbound:

Lemma 6. *Let $c \geq 1$ and $C = c \cdot \exp(\frac{1-c^2}{2}) < 1$ then for any integer $N \geq 1$ and real $r > 0$ we have*

$$\Pr_{\mathbf{x} \leftarrow D_{\mathbb{Z}^N, s}} \left[\|\mathbf{x}\|_2 \geq c \cdot s \cdot \sqrt{\frac{N}{2 \cdot \pi}} \right] \leq C^N.$$

Note that this implies that

$$\Pr_{\mathbf{x} \leftarrow D_{\mathbb{Z}^N, s}} \left[\|\mathbf{x}\|_2 \geq 2 \cdot r \cdot \sqrt{N} \right] \leq 2^{-N},$$

where $r = s/\sqrt{2 \cdot \pi}$. If we therefore select $\alpha, \beta \in D_{\mathbb{Z}^N, s}$, consider them as elements of $\mathcal{O}_K$, we then have, with overwhelming probability that $\|\alpha\|_2, \|\beta\|_2 \leq 2 \cdot r \cdot \sqrt{N}$. We then apply the standard inequality between the 2- and the 1-norm to deduce $\|\alpha\|_1, \|\beta\|_1 \leq 2 \cdot r \cdot N$. We then have that

$$\begin{aligned}\|\alpha \cdot \beta\|_\infty &\leq C_m \cdot \|\sigma(\alpha \cdot \beta)\|_\infty \leq C_m \cdot \|\sigma(\alpha)\|_\infty \cdot \|\sigma(\beta)\|_\infty \\ &\leq C_m \cdot \|\alpha\|_1 \cdot \|\beta\|_1 \\ &\leq 4 \cdot C_m \cdot r^2 \cdot N^2.\end{aligned}$$

D Security, Parameter Choice and Performance

In this Appendix we show that our concrete SHE scheme meets all the security requirements required by our MPC protocol, i.e. that it is an admissible cryptosystem. On the way we derive parameter settings, and finally we present some implementation results for the core operations.

Recall a cryptosystem is admissible if it meets the following requirements:

- It is IND-CPA secure.
- It has a KeyGen^* function with the required properties.
- It is (B_{plain}, B_{rand}, C)-correct, where $B_{plain} = N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8}$, $B_{rand} = d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8}$, and where C, the set of functions we can evaluate on ciphertexts, contains all formulas evaluated in the protocol $\Pi_{\text{PREP}'}$ (including the identity function). Note that here we choose the values for B_{plain}, B_{rand} that correspond to the most efficient variant of the ZK proofs.

Recall in the expressions for B_{plain} and B_{rand} we have d is the dimension of the randomness space, i.e. $d = 3 \cdot N$, τ is a bound on the infinity norm of valid plaintexts, i.e. $p/2$; and ρ is a bound on the infinity norm of the randomness in validly generated ciphertexts, i.e. $\rho \approx 2 \cdot r \cdot \sqrt{N}$, by the tailbound of Lemma 6.

IND-CPA and KeyGen*'s properties: We first turn to discussing security. Since our scheme is identical (bar the distributed decryption functionality) to that of [7], security can be reduced to the hardness of the following problem.

Definition 2 (PLWE Assumption). *For all* $\mathsf{sec} \in \mathbb{N}$*, let* $f(X) = f_{\mathsf{sec}}(X) \in \mathbb{Z}[X]$ *be a polynomial of degree* $N = N(\mathsf{sec})$*, let* $q = q(\mathsf{sec}) \in \mathbb{Z}$ *be a prime integer, let* $R = \mathbb{Z}[X]/f(X)$ *and* $R = R/qR$*, and let* χ *denote a distribution over the ring* R*. The* polynomial LWE assumption $\mathsf{PLWE}_{f,q,\chi}$ *states that for any* $l = poly(\mathsf{sec})$ *it holds that*

$$\{(a_i, a_i \cdot s + e_i)\}_{i\in[l]} \approx \{(a_i, u_i)\}_{i\in[l]}$$

where s *is sampled from the distribution* χ*, and* a_i, u_i *are uniformly random in* R_q*. We require computational indistinguishability to hold given only* l *samples, for some* $l = poly(\mathsf{sec})$*.*

In particular our scheme is semantically secure if the $\mathsf{PLWE}_{\Phi_m(X),q_0,\mathsf{D}^N_\rho(s)}$-problem is hard. The hardness of the same problem also implies that the output from $\mathsf{KeyGen}()$ is computationally indistinguishable from that of $\mathsf{KeyGen}^*()$.

Thus our first task is to derive relationships between the parameters so as to ensure the first two properties of being admissible are satisfied, i.e. the PLWE problem is actually hard to solve. The basic parameters of our scheme are the degree of the associated number field $N = \phi(m)$, the standard deviation r of the used Gaussian distribution, and the modulus q. We first turn to estimating r; we do this by using the "standard" analysis of the underlying LWE problem.

We first ensure that r is chosen to avoid combinatorial style attacks. Consider the underlying LWE problem as being given by $\mathbf{s} \cdot A + \mathbf{e} = \mathbf{v}$, where $\mathbf{e}$ is the LWE error vector, and A is a random $N \times t$ matrix over $\mathbb{F}_q$. In [1] the authors present a combinatorial attack which breaks LWE in time $2^{O(\|\mathbf{e}\|^2_\infty)}$ with high probability. Since $\mathbf{e}$ is chosen by the discrete Gaussian with standard deviation r, if we pick r large enough then this attack should be prevented. Thus choosing r such that $r > 3.2$ will ensure that r is large enough to avoid combinatorial attacks, i.e. $s \geq 8$.

We now turn to the distinguishing problem, namely given $\mathbf{v}$ can we determine whether it arises from an LWE sample, or from a uniform sample. We determine a lower bound on N. The natural "attack" against the decision LWE problem is to first find a short vector $\mathbf{w}$ in the dual lattice $\Lambda_q(A^\mathsf{T})^*$ and then check whether $\mathbf{w} \cdot \mathbf{v}^\mathsf{T}$ is close to an integer. If it is then the input vector is an LWE sample, if not it is random. Thus to ensure security, following the argument in [23][Section 5.4.1], we require

$$r \geq \frac{1.5}{\|\mathbf{w}\|_2}.$$

Following the work of [14] we can estimate, for $t \gg N$, the size of the output of a lattice reduction algorithm operating on the lattice $\Lambda_q(A^\mathsf{T})^*$. In particular if the algorithm tries to find a vector with root Hermite factor δ (thus δ measures the difficulty in breaking the underlying SHE system, typically one may select $\delta \approx 1.005$, but see later for other choices) then we expect to find a vector $\mathbf{w}$ of size

$$\frac{1}{q}\min(q, \delta^t \cdot q^{N/t}).$$

Following the analysis of [23] the above quantity is minimized when we select $t = t' := \sqrt{N\log(q)/\log(\delta)}$. This leads us to the deduce the lower bound

$$r \geq 1.5 \cdot \max(1\ ,\ \delta^{-t'} \cdot q^{1-N/t'}).$$

Noise of a Clean Ciphertext: We now turn to determining the bound, in the infinity norm, of the value obtained in decrypting valid ciphertexts. Consider what happens when we decrypt a clean ciphertext, encrypted via $(\mathbf{c}_0, \mathbf{c}_1) = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{x}, \mathbf{r})$, with $\mathbf{r} = (\mathbf{u}, \mathbf{v}, \mathbf{w})$. This looks like a PLWE sample $(\mathbf{c}_1, \mathbf{c}_0)$ where the "noise" term, for a validly generated clean ciphertext, is given by

$$\begin{aligned}\mathbf{t} &= \mathbf{c}_0 - \mathbf{s} \cdot \mathbf{c}_1 \\ &= \mathbf{x} + p \cdot (\mathbf{e} \cdot \mathbf{v} + \mathbf{w} + \mathbf{s} \cdot \mathbf{u})\end{aligned}$$

By our estimates in Appendix C.3 we can bound the infinity norm of $\mathbf{t}$ by

$$\|\mathbf{t}\|_\infty \leq \frac{p}{2} + p \cdot \left(4 \cdot C_m \cdot r^2 \cdot N^2 + 2 \cdot \sqrt{N} \cdot r + 4 \cdot C_m \cdot r^2 \cdot N^2\right) =: Y.$$

(B_{plain}, B_{rand}, C)-correctness: Whilst IND-CPA is about security in relation to validly created ciphertexts, our distributed decryption functionality must be secure even when some ciphertexts are not completely valid. This was why we introduced the notion of (B_{plain}, B_{rand}, C)-correctness. We need to pick B_{plain} and B_{rand} so that $B_{plain} \geq N \cdot \tau \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8}$ and $B_{rand} \geq d \cdot \rho \cdot \mathsf{sec}^2 \cdot 2^{\mathsf{sec}/2+8}$. Since $B_{plain} \ll B_{rand}$ we estimate the noise term associated to such a "clean" ciphertext will be bounded by $Y' = (B_{rand}/\rho)^2 \cdot Y = 9 \cdot N^2 \cdot \mathrm{sec}^4 \cdot 2^{\mathsf{sec}+16} \cdot Y$. In our MPC protocol we only need to be able to evaluate functions of the form

$$(x_1 + \cdots + x_n) \cdot (y_1 + \cdots + y_n) + (z_1 + \cdots + z_n).$$

We can, via the results in Appendix C.3, crudely estimate the size of B, from Section 6, needed to ensure valid decryption. Our crude (over-) estimate therefore comes out as

$$\begin{aligned}B &\leq \delta_\infty \cdot (n \cdot Y') \cdot (n \cdot Y') + (n \cdot Y') \\ &\leq C_m \cdot N^2 \cdot n^2 \cdot Y'^2 + n \cdot Y' \\ &\leq C_m \cdot N^2 \cdot n^2 \cdot c_{\mathsf{sec}}^2 \cdot Y^2 + n \cdot c_{\mathsf{sec}} \cdot Y =: Z\end{aligned}$$

where $c_{\mathsf{sec}} = 9 \cdot N^2 \cdot \mathsf{sec}^4 \cdot 2^{\mathsf{sec}+8}$. We take Z as the bound, which we then need to scale by $1 + 2^{\mathsf{sec}}$ to ensure we have sufficient space to enable the distributed decryption algorithm. Hence, the value of q needs to be selected so that $Z \cdot (1 + 2^{\mathsf{sec}}) < q/2$.

So in summary we need to choose parameters such that

$$\begin{aligned}&q > 2 \cdot Z \cdot (1 + 2^{\mathsf{sec}}), \\ &r > \max\left\{3.2, 1.5 \cdot \delta^{-t'} \cdot q^{1-N/t'}\right\},\end{aligned}$$

where sec is the statistical security parameter, δ is a measure of how hard it is to break the underlying SHE scheme, and $t' = \sqrt{N \log(q)/\log(\delta)}$. This leads to a degree of circularity in the dependency of the parameters, but valid parameter sets can be found by a simple search technique.

Specific Parameter Sets: To determine parameters for fixed values of $(\mathbb{F}_{p^k})^s$ and n we proceed as follows. There are two interesting cases; one where p is fixed (i.e. $p = 2$) and one where we only care that p is larger than some bound (i.e. $p > 2^{32}$, or $p > 2^{64}$). The latter case of $p > 2^{64}$ is more interesting as such size numbers can be utilized more readily in applications since we can simulate integer arithmetic without overflow with such numbers. In addition using such a value of p means we do not need to repeat our ZKPoKs, or replicate the MACs so as to get a cheating probability of less than 2^{-40}.

Our method in all cases is to first fix p, n, sec and δ, we then search using the above inequalities for (rough) values of q and N which satisfy the inequalities above. We then search for exact values of p and N which satisfy our functional requirements on p (i.e. fixed or greater than some bound) plus N larger than the bound above, such that N is the degree of $F(X) = \Phi_m(X)$ and $F(X)$ splits into at least s factors of degree divisible by k over $\mathbb{F}_p$.

Given this precise value for N, we then return to the above inequalities to find exact values of q and r. In all our examples below we pick $n = 3$, $\mathsf{sec} = 40$, and $\delta = 1.0052$.

Example 1. We first look at $p > 2^{32}$. Our first (approximate) search reveals we need $N > 14300$, $q \approx 2^{430}$ and $r = 3.2$ (assuming $C_m \leq 2$). We then try to find an optimal value of N; this is done by taking increasing primes $p > 2^{32}$ and factoring $p - 1$. The factors of $p - 1$ correspond to values of m such that $\Phi_m(X)$ factors into $\phi(m)$ factors modulo p. So we want to find a p such that $p - 1$ is divisible by an m, so that $N = \phi(m) > 14300$. A quick search reveals candidates of

$$(p, N, m) = (2^{32} + 32043,\ 14656,\ 14657).$$

Picking m in this way will maximise the value of $s = n$, and hence allow us to perform more operations in parallel. In addition since m is prime we know, by Lemma 5, that $C_m \approx 1.2732$, thus justifying our assumption in deriving the bounds of $C_m \leq 2$.

Selecting m to be the prime 14657 in addition allows us to evaluate $s = p-1 = 14656$ runs of the triple production algorithm in parallel. The message expansion factor, given we require $N \cdot \log_2(q)$ bits to represent N elements in $\mathbb{F}_p$ is given by

$$\frac{N \cdot \log_2(q)}{N \cdot \log_2(p)} = \frac{\log_2(q)}{\log_2(p)} = \frac{430}{32} \approx 13.437.$$

Example 2. Performing the same analysis for a $p > 2^{64}$, our first naive search of parameters reveal we need an $n \approx 16700$ and $q \approx 2^{500}$. We then search for specific parameters and find $p = 2^{64} + 4867$ is pretty near to optimum, which results in a prime value of m of 16729. We find the expansion factor is given by

$$\frac{\log_2(q)}{\log_2(p)} = \frac{500}{64} \approx 7.81.$$

Example 3. We now look at the case $p = 2$ and $k = 8$, i.e. we are looking for parameters which would allow us to compute AES circuits in parallel; or more generally circuits over $\mathbb{F}_{2^8}$. Our first approximate search reveals that we need $N > 12300$, $q \approx 2^{370}$ and $r = 3.2$. So we now need to determine a value m such that

$$N = \phi(m) > 12100 \text{ and } 2^d \equiv 1 \pmod{m} \text{ and } d \equiv 0 \pmod{8}.$$

A quick search reveals candidates of

$$(m, N) = (17425, 12800)$$

since $\Phi_{17425}(X)$ factors into $s = 320$ factors of degree $d = 40$ modulo 2. Thus using this value of m we are able to work with $s = 320$ elements of $\mathbb{F}_{2^8}$ in parallel. The message expansion factor, given we require $N \cdot \log_2(q)$ bits to represent 320 elements in $\mathbb{F}_{2^8}$ is given by

$$\frac{N \cdot \log_2(q)}{8 \cdot s} = \frac{d \cdot 370}{8} = 1850.0$$

For this value of m we find $C_{17425} \approx 9.414$.

We present the following run-times we have achieved. We time the operations for encrypting and decrypting clean ciphertexts, the time to homomorphically compute $(c_x \boxtimes c_y) \boxplus c_z$, plus the time to decrypt the said result. The times are given in seconds, and in brackets we present the amortized time per finite field element. All timings were performed on an Intel Core-2 6420 running at 2.13 GHz.

Example	Enc Time (s)	Dec (Clean) Time (s)	$(c_x \boxtimes c_y) \boxplus c_z$ Time (s)	$\mathsf{Dec}_{\mathsf{sk}}((c_x \boxtimes c_y) \boxplus c_z)$ Time (s)
1	0.72 {0.00005}	0.35 {0.00002}	1.43 {0.0001}	0.72 {0.00005}
2	3.13 {0.00019}	1.54 {0.00009}	6.27 {0.0004}	3.15 {0.00018}
3	1.26 {0.00394}	0.60 {0.00188}	2.46 {0.0077}	1.23 {0.00384}

Estimating Equivalent Symmetric Security Level: The above examples were computed using the root Hermite factor of $\delta = 1.005$. Mapping this "hardness" parameter for the underlying lattice problem to a specific symmetric security level (i.e. 80-bit security, or 128-bit security) is a bit of a "black art" at present.

In [9] the authors derive an estimate for the block size needed to obtain a given root Hermite factor, assuming an efficient BKZ lattice reduction algorithm is used. They then provide estimates as to the run time needed for a specific enumeration using this block size. As an example of their analysis they estimate that a block size of 286 is needed to obtain a root Hermite factor of $\delta = 1.005$. Then they estimate that the run time needed to perform the enumeration in a projected lattice of such dimension (the key sub-procedure of the BKZ algorithm) takes time roughly between 2^{80} and 2^{175} operations. Thus a value of $\delta = 1.005$ can be considered secure; however their estimates are not precise enough to produce parameters associated with a given symmetric security level.

In [20] the authors take a different approach and simply extrapolate run times for the NTL implementation of BKZ. By looking at various LWE instances, they derive the following equation linking the expected run-time of a distinguishing attack and the root Hermite factor

$$\log_2 T = \frac{1.8}{\log_2 \delta} - 110.$$

The problem with this approach is that NTL's implementation of BKZ is very old, and hence is not state-of-the-art; on the other hand we are able to derive a direct linkage between δ and $\log_2 T$. Using this equation we find the following equivalences:

$\log_2 T$	80	100	128	196	256
δ	1.0066	1.0059	1.0052	1.0041	1.0034

Using these estimates for δ we re-run the above analysis to find approximate values for N and q in our three example applications; again assuming $n = 3$ and $\mathsf{sec} = 40$.

	$\mathbb{F}_p : p > 2^{32}$		$\mathbb{F}_p : p > 2^{64}$		$\mathbb{F}_{2^8}$	
	$N >$	$\log_2 q \approx$	$N >$	$\log_2 q \approx$	$N >$	$\log_2 q \approx$
$\delta = 1.0066$	11300	430	12900	490	9500	360
$\delta = 1.0059$	12600	430	14700	500	10900	370
$\delta = 1.0052$	14300	430	16700	500	12300	370
$\delta = 1.0041$	18600	440	21100	500	15600	370
$\delta = 1.0034$	22400	440	25500	500	18800	370

As can be seen the security parameter has only marginal impact on $\log_2 q$, and results in a doubling of the size of N as we increase from a security level of 80 bits to 256 bits. As a comparison if we, for security level 128 bits, i.e. $\delta = 1.0052$, increase the value of sec from 40 to 80 we find the following parameter sizes:

	$\mathbb{F}_p : p > 2^{32}$		$\mathbb{F}_p : p > 2^{64}$		$\mathbb{F}_{2^8}$	
	$N >$	$\log_2 q \approx$	$N >$	$\log_2 q \approx$	$N >$	$\log_2 q \approx$
$\delta = 1.0052$	18700	560	21000	630	16700	500

E Functionalities

Functionality $\mathcal{F}_{\text{RAND}}$

Random Sample: When receiving $(rand)$ from all parties, it samples a uniform $r \leftarrow \{0,1\}^u$ and outputs $(rand, r)$ to all parties.

Random modulo p: When receiving $(rand, p)$ from all parties, it samples a uniform value $e \leftarrow \mathbb{F}_{p^k}$ and outputs $(rand, e)$ to all parties.

Fig. 14. The ideal functionality for coin-flipping.

Functionality $\mathcal{F}_{\text{AMPC}}$

Initialize: On input $(init, p)$ from all parties, the functionality activates and stores the modulus p.

Rand: On input $(rand, P_i, varid)$ from all parties P_i, with $varid$ a fresh identifier, the functionality picks $r \leftarrow \mathbb{F}_{p^k}$ and stores $(varid, r)$.

Input: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with $varid$ a fresh identifier, the functionality stores $(varid, x)$.

Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x + y \mod p)$.

Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x \cdot y \mod p)$.

Output: On input $(output, varid)$ from all honest parties (if $varid$ is present in memory), the functionality retrieves $(varid, x)$ and outputs it to the environment. If the environment inputs OK then x is output to all players. Otherwise $\perp$ is output to all players.

Fig. 15. The ideal functionality for arithmetic MPC.

Functionality $\mathcal{F}_{\text{PREP}}$

Usage: We first describe two macros, one to produce $[\![\mathbf{v}]\!]$ representations and one to produce $\langle\mathbf{v}\rangle$ representations. We denote by A the set of players controlled by the adversary.

$\mathsf{Bracket}(\mathbf{v}_1,\ldots,\mathbf{v}_n,\Delta_1,\ldots,\Delta_n,\beta_1,\ldots,\beta_n)$, where $\mathbf{v}_1,\ldots,\mathbf{v}_n,\Delta_1,\ldots,\Delta_n \in (\mathbb{F}_{p^k})^s$, $\beta_1,\ldots,\beta_n \in \mathbb{F}_{p^k}$

1. Let $\mathbf{v} = \sum_{i=1}^n \mathbf{v}_i$
2. For $i = 1,\ldots,n$
   (a) The functionality computes the MAC $\gamma(\mathbf{v})_i \leftarrow \mathbf{v}\cdot\beta_i$ and sets $\gamma_i \leftarrow \gamma(\mathbf{v})_i + \Delta_i$
   (b) For every corrupt player P_j, $j \in A$ the environment specifies a share γ_i^j
   (c) The functionality sets each share γ_i^j, $j \notin A$, uniformly such that $\sum_{j=1}^n \gamma_i^j = \gamma_i$
3. The functionality sends $(\mathbf{v}_i,(\beta_i,\gamma_1^i,\ldots,\gamma_n^i))$ to each honest player P_i (dishonest players already have the respective data).

$\mathsf{Angle}(\mathbf{v}_1,\ldots,\mathbf{v}_n,\Delta,\alpha)$, where $\mathbf{v}_1,\ldots,\mathbf{v}_n,\Delta \in (\mathbb{F}_{p^k})^s$, $\alpha \in \mathbb{F}_{p^k}$

1. Let $\mathbf{v} = \sum_{i=1}^n \mathbf{v}_i$
2. The functionality computes the MAC $\gamma(\mathbf{v}) \leftarrow \alpha\cdot\mathbf{v}$ and sets $\gamma \leftarrow \gamma(\mathbf{v}) + \Delta$
3. For every corrupt player P_i, $i \in A$ the environment specifies a share γ_i
4. The functionality sets each share γ_i $i \notin A$ uniformly such that $\sum_{i=1}^n \gamma_i = \gamma$
5. The functionality sends $(0,\mathbf{v}_i,\gamma_i)$ to each honest player P_i (dishonest players already have the respective data).

Initialize: On input $(init,p,k,s)$ from all players, the functionality stores the prime p and the integers k,s. It then waits for the environment to call either "stop" or "OK". In the first case the functionality sends "fail" to all honest players and stops. In the second case it does the following:

1. For each corrupt player P_i, $i \in A$, the environment specifies a share α_i
2. The functionality sets each share α_i, $i \notin A$ uniformly
3. For each corrupt player P_i, $i \in A$, the environment specifies a key β_i
4. The functionality sets each key β_i $i \notin A$ uniformly
5. The environment specifies $\Delta_1,\ldots,\Delta_n \in (\mathbb{F}_{p^k})^s$
6. It runs the macro $\mathsf{Bracket}(\mathsf{Diag}(\alpha_1),\ldots,\mathsf{Diag}(\alpha_n),\Delta_1,\ldots,\Delta_n,\beta_1,\ldots,\beta_n)$.

Pair: On input $(pair)$ from all players, the functionality waits for the environment to call either "stop" or "OK". In the first case the functionality sends "fail" to all honest players and stops. In the second case it does the following:

1. For each corrupt player P_i, $i \in A$, the environment specifies a share $\mathbf{r}_i$
2. The functionality sets each share $\mathbf{r}_i$, $i \notin A$ uniformly
3. The environment specifies $\Delta,\Delta_1,\ldots,\Delta_n \in (\mathbb{F}_{p^k})^s$
4. It runs the macros $\mathsf{Bracket}(\mathbf{r}_1,\ldots,\mathbf{r}_n,\Delta_1,\ldots,\Delta_n,\beta_1,\ldots,\beta_n)$ and $\mathsf{Angle}(\mathbf{r}_1,\ldots,\mathbf{r}_n,\Delta,\alpha)$.

Triple: On input $(triple)$ from all players, the functionality waits for the environment to call either "stop" or "OK". In the first case the functionality sends "fail" to all honest players and stops. In the second case it does the following

1. For each corrupt player P_i, $i \in A$, the environment specifies shares $\mathbf{a}_i,\mathbf{b}_i$
2. The functionality sets each share $\mathbf{a}_i,\mathbf{b}_i$, $i \notin A$ uniformly. Let $\mathbf{a} := \sum_{i=1}^n \mathbf{a}_i$, $\mathbf{b} := \sum_{i=1}^n \mathbf{b}_i$
3. The environment specifies $\Delta_{\mathbf{a}},\Delta_{\mathbf{b}},\delta \in (\mathbb{F}_{p^k})^s$
4. It sets $\mathbf{c} \leftarrow \mathbf{a}\cdot\mathbf{b} + \delta$
5. For each corrupt player P_i, $i \in A$, the environment specifies shares $\mathbf{c}_i$
6. The functionality sets each share $\mathbf{c}_i$, $i \notin A$ uniformly with the constrain $\sum_{i=1}^n \mathbf{c}_i = \mathbf{c}$
7. The environment specifies $\Delta_{\mathbf{c}} \in (\mathbb{F}_{p^k})^s$
8. It runs the macros $\mathsf{Angle}(\mathbf{a}_1,\ldots,\mathbf{a}_n,\Delta_{\mathbf{a}},\alpha)$, $\mathsf{Angle}(\mathbf{b}_1,\ldots,\mathbf{b}_n,\Delta_{\mathbf{b}},\alpha)$, $\mathsf{Angle}(\mathbf{c}_1,\ldots,\mathbf{c}_n,\Delta_{\mathbf{c}},\alpha)$.

Fig. 16. The ideal functionality for making the global key $[\![\alpha]\!]$, pairs $[\![\mathbf{r}]\!],\langle\mathbf{r}\rangle$ and triples $\langle\mathbf{a}\rangle,\langle\mathbf{b}\rangle,\langle\mathbf{c}\rangle$

Implementing AES via an Actively/Covertly Secure Dishonest-Majority MPC Protocol

I. Damgård[1], M. Keller[2], E. Larraia[2], C. Miles[2], and N.P. Smart[2]

[1] Department of Computer Science,
University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N,
Denmark.
[2] Dept. Computer Science,
University of Bristol,
Woodland Road,
Bristol, BS8 1UB,
United Kingdom.

Abstract. We describe an implementation of the protocol of Damgård, Pastro, Smart and Zakarias (SPDZ/Speedz) for multi-party computation in the presence of a dishonest majority of active adversaries. We present a number of modifications to the protocol; the first reduces the security to covert security, but produces significant performance enhancements; the second enables us to perform bit-wise operations in characteristic two fields. As a bench mark application we present the evaluation of the AES cipher, a now standard bench marking example for multi-party computation. We need examine two different implementation techniques, which are distinct from prior MPC work in this area due to the use of MACs within the SPDZ protocol. We then examine two implementation choices for the finite fields; one based on finite fields of size 2^8 and one based on embedding the AES field into a larger finite field of size 2^{40}.

1 Introduction

The invention of secure multi-party computation is one of the crowning achievements of theoretical cryptography, yet despite being invented around twenty-five years ago it has only recently been implemented and tested in practice. In the last few years a number of MPC "systems" have appeared [4, 7–9, 12, 15, 22], as well as experimental research results [13, 16, 21, 25, 26].

The work (both theoretical and practical) can be essentially divided into two camps. On one side we have techniques based on Yao circuits [28], which are mainly focused on two party computations, and on the other we have techniques based on secret sharing [6, 11], which can be applied to more general numbers of players. This is rather a coarse divide as some techniques, such as that from [25], only apply in the two party case but it is based on secret sharing as opposed to Yao circuits. Following this coarse divide we can then divide work into those which consider only honest-but-curious adversaries and those which consider more general active adversaries.

As in theory, it turns out that in practice obtaining active security is a much more challenging task; requiring more computational and communication resources. All prior implementation reports to our knowledge for active adversaries have either been in the two party setting, or have restricted themselves to the multi-party setting with honest majority. In the two party setting one can adopt specialist protocols, such as those based on Yao circuits, whilst the restriction to honest majority in the multi-party setting means that cheaper information theoretic constructions can be employed. Recently, Damgård et al [14] following on from work in [5], presented an actively secure protocol (dubbed "SPDZ" and pronounced "Speedz") in the multi-party setting which is secure in the presence of dishonest majority. The paper [14] contains some simple implementation results, and extrapolated estimates, but it does not report on a fully working implementation which computes a specific function.

Whilst active security is the "gold standard" of security, many applications can accept a weaker notion called covert security [1, 2]. In this model a dishonest party deviating from the protocol will be detected with high probability; as opposed to the overwhelming probability required by active security. Due to the weaker requirements, covert security can often be achieved for less computational effort.

Our Contribution. As already remarked much progress has been made on implementation of MPC protocols in the last few years, but most of the "fast" implementations have been for simpler security models. For example prior work has focused on protocols for two party computation only, or honest-but-curious adversaries only, or for threshold adversaries only. In this work we extend the prior implementation work to the most complex setting namely covert and active security against a dishonest majority. In addition we examine more than four players; with some experiments being carried out with ten players. Thus our work shows that even such stringent security requirements and parameter settings are beginning to be within reach of practical application of MPC technology.

More concretely, we show how to simplify the SPDZ protocol so that it achieves covert security for a greatly improved computational performance, we present the first implementation results for the SPDZ protocol (in both the active and covert cases), and we describe an evaluation of the AES functionality with this protocol. Our protocol implementation is in the random oracle model, specifically the zero-knowledge proofs required by SPDZ are implemented using the Fiat–Shamir heuristic. We also simplify some other parts of the SPDZ protocol in the random oracle model (details are provided below), and present extensions to enable bit-wise operations in characteristic two fields.

Since the work of [26] it has become common to measure the performance of an MPC protocol with the time it takes to evaluate the AES functionality. This is for a number of reasons: Firstly AES provides a well understood function which is designed to be highly non-linear, secondly AES has a regular and highly mathematical structure which allows one to investigate various different optimization techniques in a single function, and thirdly "oblivious" evaluation of AES on its own is an interesting application which if one could make it fast enough could have practical application.

The paper is structured as follows. We start by covering details of prior work on using MPC to implement AES. In Section 3 we detail the basics of the SPDZ protocol and the minor changes we made to the presentation in [14]. Then in Section 4 we describe how we implemented the S-Box, this is the only non-linear component in AES and so it is the only part which requires interaction. Finally in Section 5 we present our implementation results.

2 Prior Work on Evaluating AES via MPC Protocols

As noted earlier the first MPC evaluation of the AES functionality was presented in [26]. This paper presented a protocol for the case of two parties, using Yao circuits as the basic building block. On their own Yao circuits only provide security against semi-honest adversaries, and in this case the authors obtained a run-time of 7 seconds to evaluate a single AES block (the model being that party A holds the key, and party B holds a message, with B wishing to obtain the encryption of their message under A's key). To obtain security against active adversaries a variant of the cut-and-choose methodology of Lindell and Pinkas [20] was used, this resulted in the run-time dropping to 19 minutes to evaluate an AES encryption.

In [15] Henecka et al again look at two-party computation based on Yao circuits, but restrict to the case of semi-honest adversaries only. They reduce the run time per block from the previous 7 seconds down to 3.3 seconds. Huang et al [16] improve this even further obtaining a time of 0.2 seconds per block for semi-honest adversaries.

In [25] the authors present a two party protocol, but instead of their protocol being based on Yao circuits they instead base it on OT extension in the Random Oracle Model, and a form of "secret sharing with MACs" (similar to the SPDZ protocol which we examine below). This enables the authors to obtain active security and to improve on the prior performance of other implementations. The run time for a single evaluation of the AES circuit is 64 seconds, however this drops to around 2.5 seconds when amortized over a number of encryption blocks.

The most recent result in the two party setting is [17], which returns to using Yao circuit based protocols. By use of clever engineering of the overall run-time design the authors are able to significantly improve the execution time for a single AES evaluation down to 1s in the case of active adversaries.

Moving to the case of more than two players, all prior implementation results have either been for three or four players; and have been in the semi-honest setting for the case of three players. Like our work, in this setting one utilizes secret sharing but prior work has been based on Shamir secret sharing, or specialised protocols; and in the case of active security has been based on Verifiable Secret Sharing.

The main paper which is related to our work is that of [13], so we now spend some time to explain the differences between our approach and that of [13]. In [13] the authors examine an AES implementation in the case of standard threshold-secret-sharing based MPC protocols. An implementation for one semi-honest adversary amongst three players and one active adversary amongst four players is described using the VIFF framework [12]. The VIFF framework works much like the SPDZ protocol, in that it utilizes Beaver's [3] method for MPC evaluation. In an Offline Phase "multiplication triples" are produced, and then in an Online Phase the function specific calculation is performed. The two key differences between the protocol in [13] and the use of SPDZ is that the method to produce the triples is different, and the method to ensure non-cheating adversaries during the evaluation of the circuit is also different. These differences are induced since [13] is interested in threshold adversary structures, whereas we are interested in the more challenging case of dishonest majority.

The protocol of [13] is however similar to our work in that it looks at the AES circuit as a circuit over the finite field $\mathbb{F}_{2^8}$, and not as an arbitrary binary circuit. The S-Box in AES is (usually) composed of two operations an inversion in the field $\mathbb{F}_{2^8}$ followed by a linear operation on the bits of the resulting element. In [13] the authors discuss various techniques for computing the inversion, and for the bitwise linear operation they utilize a trick of bit-decomposition of the shared value. This bit-decomposition is itself implemented using the technique of pseudorandom secret sharing (PRSS) of bits.

For MPC protocols based on Shamir secret sharing, obtaining a PRSS is relatively straight forward, indeed it is a local operation assuming some set-up. However, for protocols using secret sharing with MACs (as in our approach) it is unknown how to build a PRSS in such a clean way. Thus we produce such shared random bits by executing another stage in the Offline Phase of the SPDZ protocol. We also present a simplification of the technique in [13] to use such bit-decompositions to implement the S-Box. This approach does however assume that the Offline Phase somehow "knows" that the computed function will required shared random bits; which defeats the point of having a function independent Offline stage and also adds to the run time of the Offline stage. Thus we also present a distinct approach which utilizes a surprising algebraic formulation of the S-Box.

The implementation of [13] required less than 2 seconds per AES block (including key expansion) when computing with three players and at most one semi-honest adversary, and less than 7 seconds per AES block when computing with four players and at most one active adversary. These times include the time for the Offline Phase. If one is only interested in the Online Phase times, then the active adversary case can be executed in between three and four seconds per AES block.

More recent work has focused on the case of semi-honest adversaries and three players only. Two recent results [18, 19] have used an additive secret sharing scheme and a novel multiplication protocol to perform semi-honest three party MPC in the presence of at most one adversary. In [18] the authors present an AES implementation using a novel implementation of the S-Box component via an MPC table-lookup procedure. They report being able to perform 67 AES block cipher evaluations per second. In [19] the authors report on an implementation of AES, using the Sharemind framework [7], in which they can accomplish over one thousand AES block cipher evaluations per second.

In summary Table 1 summarizes the different performance figures and security models for prior work on implementing AES using multi-party computation, with also a comparison with our own work. Like all network based protocols a significant time can be spent waiting for data, thus authors have found that executing many calculations in parallel (as in for example AES-CTR mode) can have significant performance enhancements. Thus for papers which report such results we give the improved amortized costs for multiple executions (or just the blocks-per-second count for a single execution if no improvement via amortization

occurs). However, single execution costs are still important since this deals with the case of (for example) AES-CBC mode. In our implementation we found little gain in performing multiple AES evaluations in parallel.

Paper	Security	Total Number Parties	Max Number Adv.	Time for single AES Block	(Amortized) Blocks per Sec	Expanded Key	Notes
[26]	semi-honest	2	1	7.0s	0.1	N	Yao
[15]	semi-honest	2	1	3.3s	0.3	N	Yao
[16]	semi-honest	2	1	0.2s	5.0	Y	Yao
[13]	semi-honest	3	1	1.2s	0.9	N	Shamir
[18]	semi-honest	3	1	N/A	67	Y	Additive
[19]	semi-honest	3	1	1.0s	1893	Y	Additive
[26]	covert	2	1	95s	≈ 0	N	Yao
This work	covert	2	1	0.17s	10.3	Y	SPDZ
This work	covert	3	2	0.19s	9.6	Y	SPDZ
This work	covert	4	3	0.18s	9.2	Y	SPDZ
This work	covert	5	4	0.19s	7.4	Y	SPDZ
This work	covert	10	9	0.23s	5.2	Y	SPDZ
[26]	active	2	1	19m	≈ 0	N	Yao
[25]	active	2	1	4.0s	32	N	OT
[17]	active	2	1	1.0s	1.0	Y	Yao
[13]	active	4	1	2.1s	0.5	N	Shamir
This work	active	2	1	0.26s	5.0	Y	SPDZ
This work	active	3	2	0.29s	4.7	Y	SPDZ
This work	active	4	3	0.32s	4.6	Y	SPDZ
This work	active	5	4	0.34s	4.4	Y	SPDZ
This work	active	10	9	0.41s	3.6	Y	SPDZ

Table 1. A comparison of different MPC implementations of AES. We only give the online-times for those protocols which have a pre-processing phase. We also note whether the implementation assumes a pre-expanded key or not.

In interpreting the table one needs to note that Yao based experiments usually implement a different functionality. Namely, the circuit constructor is the player holding the key. Whether the key is expanded or not refers to whether the garbled circuit has this key hardwired in or not.

3 The SPDZ Protocol

We now give an overview of the SPDZ protocol, for more details see [14]. The reader should however note we make a number of minor alterations to the basic protocol, all of which are describe below. Some of these alterations are due to us working in the random oracle model (which enables us to simplify a number of sub-protocols), whilst some are simply a functional change in terms of how inputs to the parties are created and distributed. In addition we describe how to simplify the SPDZ protocol to the case of covert adversaries.

The SPDZ protocol, being based on the Beaver circuit randomization technique [3], comes in two phases. In the first phase a large number of random triples are produced, such that each party holds a share of the triple, and such that the underlying values in the triple satisfy a multiplicative relation. This phase is referred to as the "Offline Phase" since the triples do not depend on either the function to be evaluated (bar their number should exceed a constant multiple of the number of multiplication gates in the evaluated function), and the triples do not depend on the inputs to the function to be evaluated. In the second phase, called the "Online Phase" the triples are used to evaluate the function on the given input.

The key to understanding the SPDZ protocol is to note that all values are shared with respect to a non-standard secret sharing scheme, which incorporates a MAC value. To describe this secret sharing scheme we fix a finite field $\mathbb{F}_q$. The MAC keys are values $\alpha_j \in \mathbb{F}_q$ for $1 \leq j \leq n_{\mathsf{MAC}}$ such that player i holds the share $\alpha_{j,i} \in \mathbb{F}_q$ where

$$\alpha_j = \alpha_{j,1} + \cdots + \alpha_{j,n}.$$

The shared values are then given by the following sharing of a value $a \in \mathbb{F}_q$,

$$\langle a \rangle := (\delta, (a_1, \ldots, a_n), (\gamma_{j,1}, \ldots, \gamma_{j,n})_{j=1}^{n_{\mathsf{MAC}}}),$$

where a is the shared value, δ is public and we have the equalities

$$\begin{aligned} a &= a_1 + \cdots + a_n, \\ \alpha_j \cdot (a + \delta) &= \gamma_{j,1} + \cdots + \gamma_{j,n} \text{ for } 1 \leq j \leq n_{\mathsf{MAC}}. \end{aligned}$$

Given this data representing a shared value a each player P_i holds the data $(\delta, a_i, \{\gamma_{j,i}\}_{j=1}^{n_{\mathsf{MAC}}})$. To ease notation we write $\gamma_{j,i}(a)$ to denote the share of the jth MAC on item a held by party i. Arithmetic in this representation is componentwise, more precisely we have

$$\langle a \rangle + \langle b \rangle = \langle a + b \rangle, \quad e \cdot \langle a \rangle = \langle e \cdot a \rangle \quad \text{and} \quad e + \langle a \rangle = \langle e + a \rangle,$$

where

$$e + \langle a \rangle = (\delta - e, (a_1 + e, a_2, \ldots, a_n), (\gamma_{j,1}, \ldots, \gamma_{j,n})_{j=1}^{n_{\mathsf{MAC}}}).$$

The simplicity of the above method for adding a constant value to $\langle a \rangle$ is the reason of the public value δ. In [14] the presentation is simplified to having only $n_{\mathsf{MAC}} = 1$, however the case of more general values of n_{MAC} is discussed. In our implementation having $n_{\mathsf{MAC}} > 1$ will be vital to ensure active security when dealing with small finite fields, thus we present the more general case above.

The SPDZ protocol can tolerate active adversaries and dishonest majority (ignoring the case where one of the dishonest players aborts) amongst a total of n parties. Thus we can assume that $n-1$ of the parties are dishonest and will arbitrarily deviate from the protocol. The SPDZ protocol guarantees that if the protocol terminates then the honest parties know that their resulting output is correct, except with a negligible probability. For active adversaries we set this probability, to mirror the choice in [14], to 2^{-40}. For covert adversaries we adapt the protocol so that the probability that a cheating adversary will be detected is lower bounded by

$$\min\left\{1 - q^{-n_{\mathsf{MAC}}}, 1 - q^{-n_{\mathsf{SAC}}}, \frac{1}{2 \cdot (n-1)}\right\},$$

where n_{MAC} and n_{SAC} are parameters to be discussed later and $\mathbb{F}_q$ is the finite field over which our triples are defined.

3.1 Offline Phase

The Offline Phase makes use of a somewhat homomorphic encryption (SHE) scheme, with a distributed decryption procedure, and zero-knowledge proofs. In our implementation we use the optimized non-interactive zero-knowledge proofs of knowledge (NIZKPoKs) derived from the Fiat–Shamir heuristic which are described in [14]. Thus our Offline Phase is only secure in the Random Oracle model.

The specific SHE scheme used is a variant of the BGV scheme [10] over the mth cyclotomic field. We thus have lattices of dimension $\phi(m)$, over a modulus of size Q. Each ciphertext consists of two (or three) polynomials modulo Q of degree less than $\phi(m)$. The underlying plaintext space can hold an element of $(\mathbb{F}_q)^\ell$.

The Offline Phase produces many triples of such sharings $\langle a \rangle, \langle b \rangle, \langle c \rangle$ such that $c = a \cdot b$, where these values are authenticated via a global set of n_{MAC} shared MAC keys as described above. The NIZKPoKs mentioned above have soundness error $1/2$, and so in [14], we "batch" together sec executions so as to reduce

the soundness error to $2^{-\mathsf{sec}}$. This batching, combined with the vectoral plaintext space, means that a single execution of the Offline phase produces $\mathsf{sec} \cdot \ell$ triples.

We can trivially modify the Offline Phase so that it also outputs, for characteristic two fields, a set of shared random bits and their associated MACs. We can produce one such shared bit for roughly one third of the cost of one shared triple. As for the shared triples, each invocation of the method to produce shared random bits will produce $\mathsf{sec} \cdot \ell$ bits in one go.

The main cost of the Offline phase is in the production and verification of the zero-knowledge proofs. For n players, for each proof that a player needs to produce he will need to verify $n-1$ proofs of the other players. For the case of covert adversaries we simplify the Offline Phase as follows. We do not batch together proofs, i.e. we take $\mathsf{sec} = 1$, which results in soundness error for each proof of $1/2$. In addition each player when it receives $n-1$ proofs from all other players only verifies a random proof. This means that a cheating player will be detected with probability at least $1/(2 \cdot (n-1))$ in the Offline phase, as opposed to $1 - 2^{-40}$ when we use the standard actively secure Offline Phase.

3.2 Online Phase

Given that our Offline Phase is given in the Random Oracle Model we alter the Online Phase from [14] so that it too utilizes Random Oracles. This means we can present a more efficient Online Phase than that used in [14]. Our Online Phase makes use of three hash functions: The first one H_1 is used to ensure that broadcast has happened, for this hash function we require it is one which supports an API of standard hash functions consisting of $\mathsf{Init}, \mathsf{Update}$ and $\mathsf{Finalise}$ methods. The second hash function H_2 is used to generate random values for checking the linear MAC equations and the triples. The third hash function H_3, which we model as a random oracle, is used to define a commitment scheme as follows: To commit to a value x, which we denote by $\mathsf{Commit}(x)$, one generates a random value $r \in \{0,1\}^{\mathsf{sec}}$, for some security parameter sec, and computes $\mathsf{comm} = H_3(x \| r)$. To open $\mathsf{Open}(\mathsf{comm}, x, r)$ one verifies that $\mathsf{comm} = H_3(x \| r)$ returning x if this is true, and $\perp$ if it is not.

The first change we make is in how we guarantee that consistent broadcast occurs. For the Online phase we assume that the point-to-point links between the parties are authenticated, but we need to guarantee that a dishonest party is not allowed to send different messages to different players when he is required to broadcast a single value to all players. This is done by modifying the notion of a "partial opening" from [14] and the notion of "broadcast". The "broadcasts" are ensured to be correct via the parties maintaining a hash of all values received. This is checked before the output is reconstructed; thus in the final broadcast to recover the output we utilize the re-transmit method from [14] to check consistency of the final broadcast.

In the original protocol "partial opening" just means a broadcast of the share of a value held by a party, but not the broadcast of the share of the MAC on that value. Thus only the value is opened, not the MAC on the value. However, we each ensure player maintains the running totals of the linear equations they will eventually check. In [14] these linear equations were of the form $\sum_k e^k a_k$, for some random agreed value e. This gives an error probability of T/q, where T is the number of partial openings in an execution of the Online Phase. For small values of q this is not effective, thus we replace the values e^k by the output of hash function H_2. In Figure 1 we describe our modified partial opening, and broadcast protocol, which maintains a hash value of all values broadcast; as well as a method for checking consistency.

In the Online Phase the key issue is that the triples produced by the Offline Phase may not satisfy the relation $c = a \cdot b$, nor may the MACs verify. This is because we do not ensure that the dishonest parties were "well behaved" in the Offline Phase. Thus these two properties must be checked. The Online Protocol of [14] does this as follows: To check that $c = a \cdot b$ for the triples, we will use for the MPC evaluation we "sacrifice" a set of n_{SAC} extra triples per evaluated triple. For the sacrificing method in our implementation, we adopted the naïve method of [14]. This results in consuming more triples, but is simpler computationally. To check the MAC values a series of n_{MAC} linear equations are checked at the end of the Online Phase.

Each triple sacrifice and MAC equation check can be made to hold by the adversary with probability $1/q$. Thus to reduce this to something negligible we sacrifice many triples, and utilize many MAC equations. But in the case of covert adversaries we select $n_{\mathsf{MAC}} = n_{\mathsf{SAC}} = 1$, and so the probability of a cheating adversary being detected is bounded from below by $1 - 1/q$.

$\mathsf{Init}()$**:** We initialize the following data:

1. Party i executes $H_1.\mathsf{Init}()$.
2. Party i sets $\mathsf{cnt}_i = 0$.
3. For $j = 1, \ldots, n_{\mathsf{MAC}}$
   (a) Party i sets $\hat{a}_{j,i} = 0$ and $\gamma_{j,i} = 0$.
4. Party i generates a random value $\mathsf{seed}_i \in \{0,1\}^{\mathsf{sec}}$ and sends it to all other players.

$\mathsf{Broadcast}(v_i)$**:** We broadcast v_i and receive the equivalent broadcasts from other players:

1. Party i sends v_i to each player.
2. On receipt of $\{v_1, \ldots, v_n\} \setminus \{v_i\}$ execute $H_1.\mathsf{Update}(v_1 \| \ldots \| v_n)$.
3. Return $\{v_1 + \cdots + v_n\}$.

$\mathsf{PartialOpen}(\langle a \rangle)$**:** Party i obtains the partial opening of the shared value and updates their partial sums:

1. Execute $\{a_1, \ldots, a_n\} = \mathsf{Broadcast}(a_i)$.
2. $a = a_1 + \cdots + a_n$.
3. $(e_1 \| \ldots \| e_{n_{\mathsf{MAC}}}) = H_2(0 \| \mathsf{seed}_1 \| \ldots \| \mathsf{seed}_n \| \mathsf{cnt}_i) \in \mathbb{F}_q$.
4. $\mathsf{cnt}_i = \mathsf{cnt}_i + 1$
5. For $j = 1, \ldots, n_{\mathsf{MAC}}$
   (a) $\hat{a}_{j,i} = \hat{a}_{j,i} + e_j \cdot (a + \delta_a)$.
   (b) $\gamma_{j,i} = \gamma_{j,i} + e_j \cdot \gamma_{j,i}(a)$.
6. Return a.

$\mathsf{Verify}()$**:** We check all broadcasts have been consistent:

1. Party i computes $h_i = H_1.\mathsf{Finalise}()$ and sends h_i to each player.
2. On receipt of h_j from player j, if $h_i \neq h_j$ then abort.

Fig. 1. Methods for Partial Opening and Broadcast for Party i

Both of these checks require that the parties agree on some global random values at different points in the protocol. In [14] these extra shared values are determined in the Offline Phase, via a different form of secret sharing; with the sharings being opened at the critical point in the Online protocol. The benefit of this approach is that one obtains a protocol which is UC secure without the need for Random Oracles; however the down-side is that the Offline Phase becomes relatively complex. In our work we take the view that since Random Oracles have been used in the Offline Phase one might as well exploit them in the Online Phase. Thus these shared values are obtained via a Random Oracle based commitment scheme as we now describe.

The next alteration we make to the Online Phase of [14] is that we assume that the players shares of the input values are "magically distributed" to them. This can be justified in two ways. Firstly we are only interested in timing the main Offline and Online Protocol and the input distribution phase is just an added complication. Secondly, a key application scenario for MPC is when the players are computing a function on behalf of some client. In such a situation the players do not themselves have any input, it is the client which has input. In such a situation the players would obtain their respective input shares directly from the client; thus eliminating the need entirely for a special protocol to deal with obtaining the input shares.

Our final alteration is that we utilize a new online operation, in addition to local addition and multiplication, called BitDecomposition. We first note that we can given a sharing $\langle a \rangle$ of a finite field element $a \in \mathbb{F}_{2^k} = \mathbb{F}_2[X]/F(X)$, and a set of k randomly shared bits $\langle r_i \rangle$ for $i = 0, \ldots, k-1$. Suppose we write a as $\sum_{i=0}^{k-1} a_i \cdot X^i$, our goal is to produce $\langle a_i \rangle$. Firstly via a local operation we compute a sharing of $r = \sum r_i \cdot X^i$ by computing $\langle r \rangle = \sum \langle r_i \rangle \cdot X^i$. Then we produce a masked value of a, via $\langle c \rangle = \langle a \rangle + \langle r \rangle$. The value of $\langle c \rangle$ is then opened to reveal c and we compute the decomposition $c = \sum c_i \cdot X^i$. Then we can locally compute $\langle a_i \rangle = c_i + \langle r_i \rangle$. Note, if a is known to be in a subfield of $\mathbb{F}_{2^k}$, as it will be in one of our implementations for $k = 40$, we can utilize the embedding of the subfield into the larger field to reduce the number of shared random bits needed for this decomposition down to the degree of the subfield. We refer to Appendix A for more details.

Given these alterations to the Online Phase of [14] we present the modified protocol in Figure 2 of the Appendix.

4 S-Box Implementation

We present two distinct methodologies to implement the S-Box. The first requires the Offline Phase to only produce multiplication triples, and utilizes the algebraic properties of the S-Box. The second requires the Offline Phase to also produce sharings (and associated MACs) of random bits.

4.1 S-Box Via Algebraic Operations

A key design criteria of any block cipher is that it should be highly non-linear. In addition it should be hard to write down a series of simple algebraic equations to describe the cipher. Since such equations could give rise to an attack via algebraic cryptanalysis. Indeed one reason for choosing AES as an example benchmark for MPC protocols, is that being a block cipher it should be highly non-linear and hence a challenge for MPC protocols. However, as was soon realised after the standardization of AES the S-Box (the only non-linear component in the entire cipher) can be represented in a relatively clean algebraic manner.

Our algebraic method to implement the S-Box operation is based on the analysis of AES of Murphy and Robshaw [23]. In this work the authors demonstrate that actually AES can be described by (relatively simple) algebraic formulae over $\mathbb{F}_{2^8}$, in other words the transform between byte-wise and bit-wise operations in the standard representation of the AES S-Box is a bit of a MacGuffin.

Recall the AES S-Box consists of an inversion in $\mathbb{F}_{2^8}$ (which is indeed a highly non-linear function) followed by a linear operation over the bits of the result. This is usually explained that the mixture of the two operations in two distinct finite fields "breaks any algebraic structure". This was shown to be false in [23]. Indeed one can express the S-Box calculation via the following simple polynomial

$$\begin{aligned}\text{S-Box}(z) = \text{0x63} + \text{0x8F} \cdot z^{127} + \text{0xB5} \cdot z^{191} + \text{0x01} \cdot z^{223} + \text{0xF4} \cdot z^{239} \\ + \text{0x25} \cdot z^{247} + \text{0xF9} \cdot z^{251} + \text{0x09} \cdot z^{253} + \text{0x05} \cdot z^{254}.\end{aligned}$$

where (as is usual) operations are in the finite field defined by $\mathbb{F}_{2^8} = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$ and the notation 0x12 represents the element defined by the polynomial $x^4 + x$. That the operation can be defined by a polynomial of degree bounded by 255 is not surprising, since by interpolation any functions from $\mathbb{F}_{2^8}$ to $\mathbb{F}_{2^8}$ can be represented in such a way. What is surprising is that the polynomial is relatively sparse, however this can be easily shown from first principles.

Lemma 1. *The AES S-Box can be represented by a polynomial which has a non-zero coefficient for the term i if and only if $i \in \{0, 127, 191, 223, 239, 247, 251, 253, 254\}$.*

Proof. Recall the AES S-Box consists first of inversion $z \to z^{-1} = y$ followed by an $\mathbb{F}_2$ linear operation $\mathbf{w} = A \cdot \mathbf{y}^{\mathsf{T}} + \mathbf{b}$ on the bits of the result, where $\mathbf{y}$ are the bits in y. The bit matrix A and the bit vector $\mathbf{b}$ are fixed. The final result is obtained by forming the dot-product of the $(\mathbb{F}_2)^8$ vector $\mathbf{w}$ with the fixed vector $\mathbf{x} = (1, x, x^2, x^3, x^4, x^5, x^6, x^7) \in (\mathbb{F}_{2^8})^8$.

First note that inversion in $\mathbb{F}_{2^8}$ can be accomplished by computing $z^{-1} = z^{254}$, since $z^{255} = 1$ for all $z \neq 0$. The AES standard "defines" $0^{-1} = 0$, and so the formula of z^{254} can be applied even when $z = 0$ as well.

We then note that extracting the bits $\mathbf{y} = (y_0, \ldots, y_7) \in (\mathbb{F}_2)^8$ of an element $y = y_0 + y_1 \cdot x + \cdots + y_7 \cdot x^7$ can be obtained via a linear operation on the action of Frobenius on y. This follows since Frobenius acts as a linear map, and hence by applying Frobenius eight times we find eight linear equations linking the set $\{y_0, \ldots, y_7\}$ with the Frobenius actions on y. This in turn allows us to solve for the bits $\mathbf{y} = (y_0, \ldots, y_7)$. Thus there is matrix $B \in (\mathbb{F}_{2^8})^{8\times 8}$ such that

$$\mathbf{y} = B \cdot (y, y^2, y^4, y^8, y^{16}, y^{32}, y^{64}, y^{128})^{\mathsf{T}}.$$

Hence, the output of the S-Box can be written as

$$\begin{aligned}\text{S-Box}(z) &= \mathbf{x} \cdot (A \cdot \mathbf{y} + \mathbf{b}), \\ &= \mathbf{x} \cdot (A \cdot B) \cdot (y, y^2, y^4, y^8, y^{16}, y^{32}, y^{64}, y^{128})^{\mathsf{T}} + \mathbf{x} \cdot \mathbf{b}, \\ &= \mathbf{s} \cdot (1, y, y^2, y^4, y^8, y^{16}, y^{32}, y^{64}, y^{128})^{\mathsf{T}}\end{aligned}$$

where $\mathbf{s}$ is a fixed nine dimensional vector over $\mathbb{F}_{2^8}$. On replacing y with z^{254} in the above equation, using $z^{255} = 1$ for all $z \neq 0$, we obtain our result. With the result also following for $z = 0$ by inspection.

Finally to implement the S-Box we therefore need an efficient method to obtain from an shared input value z, the shared values of the elements $\{z^{127}, z^{191}, z^{223}, z^{239}, z^{247}, z^{251}, z^{253}, z^{254}\}$. This is equivalent to finding a short addition chain for the set $\{127, 191, 223, 239, 247, 251, 253, 254\}$. We found the shortest such addition chain consists of eighteen additions and is the chain

$$\{1, 2, 3, 6, 12, 15, 24, 48, 63, 64, 96, 127, 191, 223, 239, 247, 251, 253, 254\}.$$

Thus to evaluate a single S-Box requires eighteen MPC multiplication operations, as well as some local computation. Hence, to evaluate the entire AES cipher we require $18 \cdot 16 \cdot 10 = 2880$ MPC multiplications.

Looking ahead each multiplication operation will require interaction, and to reduce execution times we need to ensure that each player is kept "busy", i.e. is not left waiting for data to arrive. To do this we will interleave various different multiplications together; essentially exploiting the instruction level parallelism (ILP) within the basic AES algorithm. Clearly one can execute each of the 16 S-Box operations in a single round in parallel, thus obtaining an immediate 16-fold factor of ILP. However, further ILP can be exploited in the addition chain above as can be seen from its graphical realisation in Figure B. in the Appendix. We see that the addition chain can be executed in twelve parallel multiplication steps; thus the total number of rounds of multiplication need for the entire AES cipher will be $12 \cdot 10 = 120$.

4.2 S-Box Via BitDecomposition

As explained in [13] the S-Box can be implemented if one has access to shared random bits, via the BitDecomposition operation. In our second implementation choice we extend this technique, and reduce even further the amount of interaction needed to compute the S-Box.

We use this BitDecomposition trick in two ways. The first way is to decompose an element in $\mathbb{F}_{2^8}$ into it's bit components, so as to apply the linear map of the S-Box. This part is exactly as described in [13]; except when we open the value of $\langle c \rangle$ we perform a partial opening, leaving the checking of the MACs until the end.

In our second application of BitDecomposition we use BitDecomposition to implement the operation $x \longrightarrow x^{254}$. This done as follows: We decompose x into it's constituent bits. Then the operations $x \longrightarrow x^2$, $x \longrightarrow x^4$ are all *linear operations*, and so can be performed locally. Finally the value of $x^{254} = x^{-1}$ is computed via the combination

$$x^{254} = \left(\left(x^2 \cdot x^4\right) \cdot \left(x^8 \cdot x^{16}\right)\right) \cdot \left(\left(x^{32} \cdot x^{64}\right) \cdot x^{128}\right),$$

which requires a total of six multiplications. We could reduce this down to four multiplications by applying the Frobenius map to other elements [27]; but this will consume even more random bits per S-Box thus we settled for the above implementation which consumes 16 sharings of random bits per S-Box invocation.

5 Experimental Results

We implemented the SPDZ protocol over finite fields of characteristic two and used it to evaluate the AES function, with the S-Box implemented using both the algebraic formulation described earlier and the variant by BitDecomposition. As described earlier we examined the case of dealing with both covert adversaries and fully malicious (a.k.a. active) adversaries (with cheating probability of 2^{-40}). We note that the probability of 2^{-40} could be extended to smaller values, but we used 2^{-40} so as to be comparable with the theoretical run-time estimates given in [14]. For example to reduce the probability down to 2^{-80} would essentially require a doubling of the cost of both the Offline and Online stages.

The first decision one needs to take is as to what finite field one should work with. Since we are evaluating AES it is natural to pick the field

$$K_8 = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1).$$

Another choice, particularly suited to our active adversary cheating probability of 2^{-40}, would be to use the field

$$K_{40} = \mathbb{F}_2[y]/(y^{40} + y^{20} + y^{15} + y^{10} + 1).$$

Using this finite field has the advantage that, for active adversaries, we only need to keep one MAC share per data item, and only one triple per multiplication needs to be sacrificed. In addition the field K_8 lies in K_{40} via the embedding $x = y^5 + 1$. We also for means of comparison of the Offline phase implemented the Offline protocol over a finite field $\mathbb{F}_q$ with q a 64-bit prime.

We also experimented with various numbers of players, and different values of n_{MAC} and n_{SAC}. As explained in [14] all such variants lead to different basic parameters (m, Q, ℓ) of the underlying SHE scheme.

We now determine values of (m, Q, ℓ) for our SHE scheme given a specific finite field $\mathbb{F}_q$ (or in the case of q prime a rough size for q), a value for the sec (the number of NIZKPoKs we run in parallel in the Offline stage), and the number of players n. As a "lattice security parameter" we selected $\delta = 1.0052$ which corresponds to roughly 128 bits of symmetric security.

We require finite fields $\mathbb{F}_q$ of size $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{40}}$, as well for comparison a finite field where q was a 64-bit prime. We also looked for parameters for $n \in \{2, 3, 4, 5, 10\}$ and $\mathsf{sec} \in \{1, 40\}$. As in [14] we first search for rough estimate of the parameters (m, Q) which fit these needs:

$\mathsf{char}(\mathbb{F}_q)$	n	sec	$\phi(m) \geq$	$\log_2(Q)$
2	$2 \leq n \leq 10$	40	12300	370
2	$2 \leq n \leq 5$	1	8000	200
2	10	1	8000	210
$\approx 2^{64}$	$2 \leq n \leq 10$	40	16700	500
$\approx 2^{64}$	$2 \leq n \leq 5$	1	11000	330
$\approx 2^{64}$	10	1	11300	340

We then selected values for m as follows:

$\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{40}}$, $\mathsf{sec} = 40$: We select $m = 17425$, which gives us $\phi(m) = 12800$. The polynomial $\Phi_m(X)$ factors modulo two into $\ell = 320$ factors each of degree 40. Thus these parameters can support both our finite fields $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{40}}$.

$\mathbb{F}_{2^8}$, $\mathsf{sec} = 1$: We select $m = 13107$, which gives us $\phi(m) = 8192$. The polynomial $\Phi_m(X)$ factors modulo two into $\ell = 512$ factors each of degree 16.

$\mathbb{F}_{2^{40}}$, $\mathsf{sec} = 1$: We select $m = 13175$, which gives us $\phi(m) = 9600$. The polynomial $\Phi_m(X)$ factors modulo two into $\ell = 240$ factors each of degree 40.

$p \approx 2^{64}$, $\mathsf{sec} = 40$: We select, as in [14], $p = 2^{64} + 4867$ and $m = 16729$ so that $\ell = \phi(m) = 16728$.

$p \approx 2^{64}$, $\mathsf{sec} = 1$: We select, as in [14], $p = 2^{64} + 8947$ and $m = 11971$ so that $\ell = \phi(m) = 11970$.

Recall that one invocation of the Offline Phase produces $\mathsf{sec} \cdot \ell$ triples; thus using the choices above we obtain the following summary table, where "# Trip/# Bits" denotes the number of triples/bits produced per invocation of the Offline Phase.

Field	Adversary Type	sec	$n_{\mathsf{MAC}} = n_{\mathsf{SAC}}$	# Trip/ # Bits
K_8	covert	1	1	512
K_8	active	40	5	12800
K_{40}	covert	1	1	240
K_{40}	active	40	1	12800

We ran the Offline phase on machines with Intel i5 CPU's running at 2.8 GHz. with 4 GB of RAM. The ping between machines over the local area network was approximately 0.3 ms. We obtained the executions time given in Table 2 and Table 3, for the two different finite field choices and covert/active security choices, and various numbers of players. We did not run an example with ten players and active adversaries since this took too long. We first ran the Offline Phase in each example to produce a minimum of 5000 triples. Clearly for some parameter sets a single run produced much more than 5000, whilst for others we required multiple runs so as to reach 5000 triples. These results are in Table 2. These runs are compatible with our algebraic S-Box formulation.

This table also presents the average time needed to produce each triple, plus also the amortized time to produce triples per AES invocation (in the case where one wants to evaluate the AES functionality many times). Recall to evaluate the AES functionality with our method requires $10 \cdot 16 \cdot 18 = 2880$ multiplications in total; thus the number of triples needed is $2880 \cdot (n_{\mathsf{SAC}} + 1)$, since each multiplication consumes $n_{\mathsf{SAC}} + 1$ triples. What is clear from the table is that if one is wishing to obtain security against covert adversaries then utilizing the field K_8 is preferable. However, for security against active adversaries the field K_{40} is to be preferred.

		Covert Security			Active Security		
Field	Num. Parties	Total Time (h:m:s)	Time per Triple (seconds)	Offline time per AES blk (h:m:s)	Total Time (h:m:s)	Time per Triple (seconds)	Offline time per AES blk (h:m:s)
		No. Triples Produced: 5120			No. Triples Produced: 12800		
K_8	2	0:01:31	0.018	0:01:42	1:25:57	0.403	1:56:02
K_8	3	0:01:32	0.018	0:01:43	1:50:25	0.518	2:29:03
K_8	4	0:01:32	0.018	0:01:43	2:14:16	0.629	3:01:15
K_8	5	0:01:33	0.018	0:01:44	2:37:30	0.738	3:32:37
K_8	10	0:01:48	0.021	0:02:01	4:40:15	1.314	6:18:20
		No. Triples Produced: 5040			No. Triples Produced: 12800		
K_{40}	2	0:05:08	0.061	0:05:52	0:29:34	0.136	0:13:18
K_{40}	3	0:05:13	0.062	0:05:57	0:38:18	0.180	0:17:14
K_{40}	4	0:05:14	0.062	0:05:58	0:46:02	0.216	0:20:42
K_{40}	5	0:05:17	0.063	0:06:02	0:55:51	0.262	0:25:07
K_{40}	10	0:06:02	0.072	0:06:53	1:39:14	0.465	0:44:39

Table 2. Offline Run Time Examples For The Algebraic S-Box Method

We then run an Offline phase tailored to our BitDecomposition S-Box formulation. Here we need to perform $10 \cdot 16 \cdot 6 = 960$ multiplications, and thus we require $960 \cdot (n_{\mathsf{SAC}} + 1)$ triples to evaluate a single block. But we also require $10 \cdot 16 \cdot 16 = 2560$ shared random bits so as to perform two eight bit, BitDecompositions per S-Box invocation. Thus in Table 3 we present run times for a second invocation of the Offline Phase in which we aimed to produce a minimum of 5000 triples and 6600 shared random bits (which is the correct ratio for covert security). Due to the inbalance between Triple and Bit production the "Offline Time per AES Block" column needs to be taken as rough estimate. Again we see that for covert security K_8 is preferable, and for active security K_{40} is preferable.

But, these run times do not seem comparable with the 13ms per triple estimated by the authors of [14] for the Offline Phase. However, this discrepancy can easily be explained. The run time estimates in [14] are given for arithmetic circuit evaluation over a finite field of prime characteristic of 64-bits. With the parameter choices in [14] this means one can select parameters for the SHE scheme which enable a 16000-fold SIMD parallelism. For our finite fields of degree two the amount of SIMD parallelism in the Offline Phase is much lower than this. To see the difference that using large prime characteristic fields makes to the Offline Phase we implemented it, using the parameters above to obtain the results in Table 4. As can be seen from the

Field	Number Players	Covert Security: Total Time (h:m:s)	Covert Security: Offline Time per AES Block (h:m:s)	Active Security: Total Time (h:m:s)	Active Security: Offline Time per AES Block (h:m:s)
		No. Triples/Bits: 5120/6556		No. Triples/Bits: 12800/12800	
K_8	2	0:02:07	0:00:47	1:54:42	0:51:36
K_8	3	0:02:10	0:00:49	2:26:21	1:05:51
K_8	4	0:02:13	0:00:50	2:56:47	1:19:33
K_8	5	0:02:36	0:00:52	3:29:49	1:34:25
K_8	10	0:02:33	0:00:58	6:06:20	2:44:51
		No. Triples/Bits: 5040/6720		No. Triples/Bits: 12800/12800	
K_{40}	2	0:07:12	0:02:43	0:36:14	0:05:26
K_{40}	3	0:07:12	0:02:43	0:47:30	0:07:07
K_{40}	4	0:07:19	0:02:47	0:58:55	0:08:57
K_{40}	5	0:07:24	0:02:49	1:10:33	0:10:34
K_{40}	10	0:08:32	0:03:15	2:10:03	0:19:32

Table 3. Offline Run Time Examples For The S-Box Via BitDecomposition

table we produce triples for prime fields of 64-bits in size around twice as fast as the estimates in [14] would predict.

Number Players	Covert Security: Total Number Triples	Covert Security: Total Time (h:m:s)	Covert Security: Time per Triple (seconds)	Active Security: Total Number Triples	Active Security: Total Time (h:m:s)	Active Security: Time per Triple (seconds)
2	11970	0:00:27	0.002	669120	1:10:48	0.006
3	11970	0:00:27	0.002	669120	1:32:13	0.008
4	11970	0:00:28	0.002	669120	1:55:05	0.010
5	11970	0:00:29	0.002	669120	2:20:42	0.013
10	11970	0:00:31	0.002	669120	4:17:10	0.023

Table 4. Offline Run Time Examples For $\mathbb{F}_p$ With $p \approx 2^{64}$

We now turn to the Online Phase; recall that this itself comes in two steps (and two variants). In the first step we evaluate the function itself (consuming the triples produced in the Offline Phase), whereas in the second step we check the MAC values and open the final result. In Table 5 we present the run-times to evaluate the AES functionality for the various parameter sets generated above using our algebraic formulation of the S-Box. These are average run-times from all the players, executed over 20 different runs. The Online Phase was run on the same machines as in the Offline Phase. In Table 6 we present the same times using the S-Box variant utilizing the BitDecomposition method.

The networking between players was implemented in a point-to-point fashion with each player acting as both a server and a client. We ensured that data was sent over the sockets as soon as it was ready by disabling Nagle's algorithm [24]. To complete the function evaluation each player first parses a program written in a specialised instruction language. This allows our implementation to take advantage of the instruction level parallelism as described above so as to schedule many multiplication operations to happen in parallel.

Again we see that if security against covert adversaries is the goal then using the field K_8 is to be preferred. However, for security against active adversaries the field K_{40} performs better. We also ran the Online Phase in a run which performed ten AES encryptions in parallel. This resulted in only a small improvement in

Field	Number Players	Covert Security: Function Evaluation	Covert Security: Checking Step	Covert Security: Total Time	Active Security: Function Evaluation	Active Security: Checking Step	Active Security: Total Time
K_8	2	0.284	0.017	0.301	1.319	0.031	1.350
K_8	3	0.307	0.062	0.369	1.381	0.035	1.416
K_8	4	0.316	0.027	0.343	1.422	0.028	1.450
K_8	5	0.344	0.034	0.378	1.461	0.018	1.479
K_8	10	0.444	0.010	0.454	1.659	0.023	1.682
K_{40}	2	0.449	0.012	0.461	0.460	0.021	0.481
K_{40}	3	0.486	0.022	0.498	0.475	0.025	0.500
K_{40}	4	0.490	0.042	0.532	0.486	0.055	0.541
K_{40}	5	0.508	0.037	0.544	0.510	0.026	0.536
K_{40}	10	0.765	0.021	0.786	0.672	0.017	0.689

Table 5. Online Phase Runtime Examples (all in seconds) – Algebraic S-Box

Field	Number Players	Covert Security: Function Evaluation	Covert Security: Checking Step	Covert Security: Total Time	Active Security: Function Evaluation	Active Security: Checking Step	Active Security: Total Time
K_8	2	0.156	0.009	0.165	0.569	0.011	0.580
K_8	3	0.178	0.008	0.186	0.616	0.019	0.635
K_8	4	0.169	0.015	0.184	0.620	0.015	0.635
K_8	5	0.173	0.019	0.192	0.727	0.019	0.746
K_8	10	0.211	0.015	0.226	0.722	0.044	0.766
K_{40}	2	0.260	0.006	0.266	0.256	0.004	0.260
K_{40}	3	0.303	0.009	0.312	0.279	0.011	0.290
K_{40}	4	0.303	0.010	0.313	0.287	0.029	0.316
K_{40}	5	0.319	0.022	0.341	0.319	0.016	0.335
K_{40}	10	0.399	0.016	0.415	0.387	0.027	0.414

Table 6. Online Phase Runtime Examples (all in seconds) – S-Box Via BitDecomposition

time per AES block over executing just one AES encryption at a time, thus we do not present these figures. Improving the throughput for parallel execution is the subject of future research.

Overall, the two methods of AES evaluation are roughly comparable. The method via BitDecomposition being faster, and significantly faster when one also takes into account the associated cost of the Offline Phase. However, as remarked previously this method does not result in a generic Offline Phase; since the Offline Phase needs to "know" the expected ratio of Bits to Triples that it needs to produce for the actual function which will be evaluated in the Online Phase.

In summary we have presented the first experimental results for running MPC protocols with large numbers of players (10 as opposed to the four or less of prior work), and for a dishonest majority of active or covert adversaries (as opposed to threshold adversaries). It is expected that our reported execution times will fall as dramatically as those have done for two party MPC protocols in the last couple of years. Thus we can expect actively/covertly secure MPC protocols for dishonest majority to be within reach of some practical applications within a few years.

6 Acknowledgements

The first author acknowledges the support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of

Interactive Computation, within which [part of] this work was performed; and also from the CFEM research center (supported by the Danish Strategic Research Council) within which part of this work was performed.

The second, third and fifth author were partially supported by EPSRC via grant COED–EP/I03126X. The fifth author was also supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079, and by a Royal Society Wolfson Merit Award. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, AFRL, the U.S. Government, the European Commission or EPSRC.

References

1. Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Theoretical Cryptography Conference – TCC 2007*, Springer LNCS 4392, 137–156, 2007.
2. Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, **23**, 281–343, 2010.
3. D. Beaver, Correlated pseudorandomness and the complexity of private computations. *Symposium on Theory of Computing – STOC 1996*, ACM, 479–488, 1996.
4. A. Ben-David, N. Nisan and B. Pinkas. FairplayMP: a system for secure multi-party computation. *Computer and Communications Security – CCS 2008*, ACM, 257–266, 2008.
5. R. Bendlin, I. Damgård, C. Orlandi and S. Zakarias. Semi-homomorphic encryption and multiparty computation. *Advanced in Cryptology – EUROCRYPT 2011*, Springer LNCS 6632, 169–188, 2011.
6. M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. *Symposium on Theory of Computing – STOC 1988*, ACM, 1–10, 1988.
7. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. *European Symposium on Research in Computer Security – ESORICS 2008*, Springer LNCS 5283, 192–206, 2008.
8. P. Bogetoft, D.L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J.D. Nielsen, J.B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach and T. Toft. Secure multiparty computation goes live, *Financial Cryptography and Data Security – FC 2009*, Springer LNCS 5628, 325–343, 2009.
9. P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter. and T. Toft. A practical implementation of secure auctions based on multiparty integer computation. *Financial Cryptography and Data Security – FC 2006*, Springer LNCS 4107, 142–147, 2006.
10. Z. Brakerski, C. Gentry and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Innovations in Theoretical Computer Science –ITCS 2012*, 309–325, ACM, 2012.
11. D. Chaum, C. Crepeau and I Damgård. Multiparty unconditionally secure protocols. *Symposium on Theory of Computing – STOC 1988*, ACM, 11–19, 1988.
12. I. Damgård, M. Geisler, M. Kroigård, and J.B. Nielsen. Asynchronous multiparty computation: Theory and implementation. *Public Key Cryptography – PKC 2009*, Springer LNCS 5443, 160–179, 2009.
13. I. Damgård and M. Keller. Secure multiparty AES. *Financial Cryptography and Data Security – FC 2010*, Springer LNCS 6051, 367–374, 2010.
14. I. Damgård, V. Pastro, N.P. Smart and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. To appear *Advances in Cryptology – CRYPTO 2012*, IACR e-print 2011/535, `http://eprint.iacr.org/2011/535`, 2011.
15. W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for automating secure two-party computations. *Computer and Communications Security – CCS 2010*, ACM, 451–462, 2010.
16. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. *Proc. USENIX Security Symposium*, 2011.
17. B. Kreuter, a. shelat, and C.-H. Shen. Towards billion-gate secure computation with malicious adversaries. IACR e-print 2012/179, `http://eprint.iacr.org/2012/179`, 2012.
18. J. Launchbury, A. Adams-Moran, and I. Diatchki. Efficient lookup-table protocol in secure multiparty computation. Manuscript, 2012.
19. S. Laur, R. Talviste, and J. Willemson. AES block cipher implementation and secure database join on the SHAREMIND secure multi-party computation framework. Manuscript, 2012.

20. Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. *Advances in Cryptology – EUROCRYPT 2007*, Springer LNCS 4515, 52–78, 2007.
21. Y. Lindell, B. Pinkas, N.P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. *Security and Cryptography for Networks – SCN 2008*, Springer LNCS 5229, 2–20, 2008.
22. D. Malkhi, N. Nisan, B. Pinkas and Y. Sella. Fairplay — a secure two-party computation system. *Proc. USENIX Security Symposium*, 2004.
23. S. Murphy and M.J.B. Robshaw. Essential algebraic structure within the AES. *Advances in Cryptology – CRYPTO 2002*, Springer LNCS 2442, 1–16, 2002.
24. J. Nagle. Congestion control in IP/TCP internetworks. IETF RFC 896, 1984.
25. J.B. Nielsen, P.S. Nordholt, C. Orlandi, and S. Sheshank Burra. A new approach to practical active-secure two-party computation. IACR e-print 2011/91, `http://eprint.iacr.org/2011/91`, 2011.
26. B. Pinkas, T. Schneider, N.P. Smart, and S.C. Williams. Secure two-party computation is practical. *Advances in Cryptology – ASIACRYPT 2009*, Springer LNCS 5912, 250–267, 2009.
27. M. Rivain and E. Prouff. Provably secure higher-order masking of AES. *Cryptographic Hardware and Embedded Systems – CHES 2010*, Springer LNCS 6225, 413–427, 2010.
28. A. Yao. Protocols for secure computation. *Proc. Foundations of Computer Science – FoCS 1982*, IEEE Press, 160–164, 1982.

A Generalized BitDecomposition

In this section, we describe a generalized variant of BitDecomposition, which includes bit-decomposition in K_8 as a subfield of K_{40}.

Let $f : V \to W$ be a linear map between two vector spaces over $\mathbb{F}_2$. Then, $\langle r \rangle$ and $\langle f(r) \rangle$ for a random element $r \in V$ allows to securely compute $\langle f(x) \rangle$ for any $\langle x \rangle$ by computing and opening $\langle x + r \rangle$, and then computing $\langle f(x) \rangle = f(x + r) + \langle f(r) \rangle$.

For bit-decomposition in K_8, define $f : K_8 \to \mathbb{F}_2^8$ by

$$f\big(\sum_{i=0}^{7} a_i \cdot X^i\big) := (a_0, \ldots, a_7).$$

This function clearly is linear over $\mathbb{F}_2$. In the offline phase, it suffices to generate $\langle (r_0, \ldots, r_7) \rangle = (\langle r_0 \rangle, \ldots, \langle r_7 \rangle)$ for random bits $(r_0, \ldots, r_7)$ because $\langle r \rangle = \sum_{i=0}^{7} \langle r_i \rangle \cdot X^i$ can be computed locally. Note that $r_0, \ldots, r_7$ are understood as elements of K_8, like all variables in the protocol over K_8. Therefore, one has to make sure that they are in fact 0 or 1 and not another element of K_8. This is done by modifying the Offline Phase; in particular each party encrypts a random bit and proves that it is actually a bit. The homomorphic structure of the NIZKPoKs makes this straight-forward. As with the triples components, the secret bit is defined as the sum of all inputs, and the secret sharing with MAC is computed by multiplication via the homomorphic property of the ciphertexts and threshold decryption.

We now move to bit-decomposition for K_8 embedded in K_{40}. Let $\imath$ denote the embedding of K_8 in K_{40}. This embedding is a field homomorphism and thus a linear map between vector spaces over $\mathbb{F}_2$. The bit-decomposition for $\imath(K_8)$ is defined by $f : \imath(K_8) \to \mathbb{F}_2^8$,

$$f\big(\imath\big(\sum_{i=0}^{7} a_i \cdot X^i\big)\big) := (a_0, \ldots, a_7).$$

Again, f is linear over $\mathbb{F}_2$, and thus, the protocol explained above is applicable. Similarly to the case of K_8, it suffices to generate eight bits $(\langle r_0 \rangle, \ldots, \langle r_7 \rangle)$ in the offline phase. There is one peculiarity in this case: We defined f over $\imath(K_8) \subset K_{40}$, not K_{40}. That means, we assume that the input of f is an element of $\imath(K_8)$, not an arbitrary element. This is guaranteed in our application, but may not be true in general.

In general the function f can easily be extended to $f' : K_{40} \to \mathbb{F}_2^8$ by defining $f'(x) := f(p_{\imath(K_8)}(x))$ for $p_{\imath(K_8)}$ denoting the natural projection to $\imath(K_8)$. However, masking an arbitrary element $x \in K_{40}$ with a random element of $\imath(K_8)$ reveals $x - p_{\imath(K_8)}(x)$. Therefore, one has to mask x additionally with a random

$r' \in K_{40}/\imath(K_8)$ before opening it, i.e., compute and open $\langle x+\imath(\sum_{i=0}^{7} r_i \cdot X^i)+r' \rangle$. As above, the homomorphic structure of the NIZKPoKs allow to generate $\langle r' \rangle$ with the same cost as a random element.

The above discussion re $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{40}}$ can be extended to an arbitrary field $\mathbb{F}_{2^n}$ and a subfield $\mathbb{F}_{2^m}$ if required.

B Figures

Online Protocol

Initialize: We assume i) the parties have already invoked the Offline Phase to obtain a sufficient number of multiplication triples $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$; ii) each party holds its share of the global MAC keys $\alpha_{j,i}$; iii) that the parties have obtained (by some means) the $\langle \cdot \rangle$ sharing of the input values to the computation.

1. The parties execute $\mathsf{Init}()$ to initialize their local copy of the hash function H_1, and the values seed_i, cnt_i, $\hat{a}_{j,i}$, and $\gamma_{j,i}$.
2. The parties generate global random values $t_j \in \mathbb{F}_q$ for $j = 1, \ldots, n_{\mathsf{SAC}}$ by computing $(t_1 \| \ldots \| t_{n_{\mathsf{SAC}}}) = H_2(1 \| \mathsf{seed}_1 \| \ldots \| \mathsf{seed}_n)$.

The following steps are performed according to the circuit being evaluated.

Add: To add two representations $\langle x \rangle, \langle y \rangle$,the parties locally compute $\langle x \rangle + \langle y \rangle$.

Multiply: To multiply $\langle x \rangle, \langle y \rangle$ the parties do the following:

1. They take $n_{\mathsf{SAC}}+1$ triples $(\langle a \rangle, \langle b \rangle, \langle c \rangle), (\langle f_i \rangle, \langle g_i \rangle, \langle h_i \rangle)_{i=1}^{n_{\mathsf{SAC}}}$ from the set of the available ones (and update this latter list by deleting these triples).
2. For $j = 1, \ldots, n_{\mathsf{SAC}}$ player P_i computes
   (a) $\rho_j = \mathsf{PartialOpen}(t_j \cdot \langle a \rangle - \langle f_j \rangle)$.
   (b) $\sigma_j = \mathsf{PartialOpen}(\langle b \rangle - \langle g_j \rangle)$.
   (c) $\tau_j = \mathsf{PartialOpen}(t_j \cdot \langle c \rangle - \langle h_j \rangle - \sigma_j \cdot \langle f_j \rangle - \rho_j \cdot \langle g_j \rangle - \sigma_j \cdot \rho_j)$.
   (d) If $\tau_j \neq 0$ then abort.
3. If no player has aborted the triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ is accepted, and the parties execute $\epsilon = \mathsf{PartialOpen}(\langle x \rangle - \langle a \rangle)$ and $\delta = \mathsf{PartialOpen}(\langle y \rangle - \langle b \rangle)$.
4. The parties locally compute the answer $\langle z \rangle = \langle c \rangle + \epsilon \cdot \langle b \rangle + \delta \cdot \langle a \rangle + \epsilon \cdot \delta$

BitDecomposition: This produces the BitDecomposition of a shared value $\langle a \rangle$. We present a simplified protocol for when $q = 2^k$.

1. $c = \mathsf{PartialOpen}\left(\langle a \rangle + \sum_{i=0}^{k-1} \langle r_i \rangle \cdot X^i\right)$.
2. Write $c = \sum_{i=0}^{k-1} c_i \cdot X^i$.
3. Output $\langle a_i \rangle = c_i + \langle r_i \rangle$.

Output: We enter this stage when the players have $\langle y \rangle$ for the output value y, but this value has not yet been opened. This output value is only correct if players have behaved honestly, which we now need to check. Let $a_1, \ldots, a_T$ be all values publicly opened so far, where $\langle a_k \rangle = (\delta_k, (a_{k,1}, \ldots, a_{k,n}), (\gamma_{j,1}(a_k), \ldots, \gamma_{j,n}(a_k))_{j=1}^{n_{\mathsf{MAC}}})$.

1. Player P_i computes $(\mathsf{comm}_i, r_i) = \mathsf{Commit}(y_i \| (\gamma_{j,i}(y))_{j=1}^{n_{\mathsf{MAC}}})$.
2. The players execute $\{\mathsf{comm}_1, \ldots, \mathsf{comm}_n\} = \mathsf{Broadcast}(\mathsf{comm}_i)$.
3. For $j = 1, \ldots, n_{\mathsf{MAC}}$ the players execute
   (a) Player P_i computes $(\mathsf{comm}_{j,i}, r_{j,i}) \leftarrow \mathsf{Commit}(\gamma_{j,i})$.
   (b) Execute $\{\mathsf{comm}_{j,1}, \ldots, \mathsf{comm}_{j,n}\} = \mathsf{Broadcast}(\mathsf{comm}_{j,i})$.
   (c) Execute $\{\alpha_{j,1}, \ldots, \alpha_{j,n}\} = \mathsf{Broadcast}(\alpha_{j,i})$.
   (d) Player P_i computes $\alpha_j = \alpha_{j,1} + \cdots + \alpha_{j,n}$.
   (e) All players open $\mathsf{comm}_{j,i}$ to $\gamma_{j,i}$ (via a call to $\mathsf{Broadcast}$), the commitments are checked and if Open returns $\perp$ for a player then it aborts.
   (f) Each player verifies that $\alpha_j \cdot \hat{a}_{j,i} = \sum_i \gamma_{j,i}$ for his own values of $\hat{a}_{j,i}$.
4. The players execute $\mathsf{Verify}()$ to confirm all broadcasts have been valid.
5. To obtain the output value y, the commitments to $y_i, \gamma_{j,i}(y)$ are opened via each player transmitting to their openings to each player, and each player transmitting what it receives to each other to check consistency.
6. Now, y is defined as $y := \sum_i y_i$ and each player checks that $\alpha_j \cdot (y + \delta_y) = \sum_i \gamma_{j,i}(y)$, for $j = 1, \ldots, n_{\mathsf{MAC}}$.

Fig. 2. The (slightly) modified SPDZ online phase.

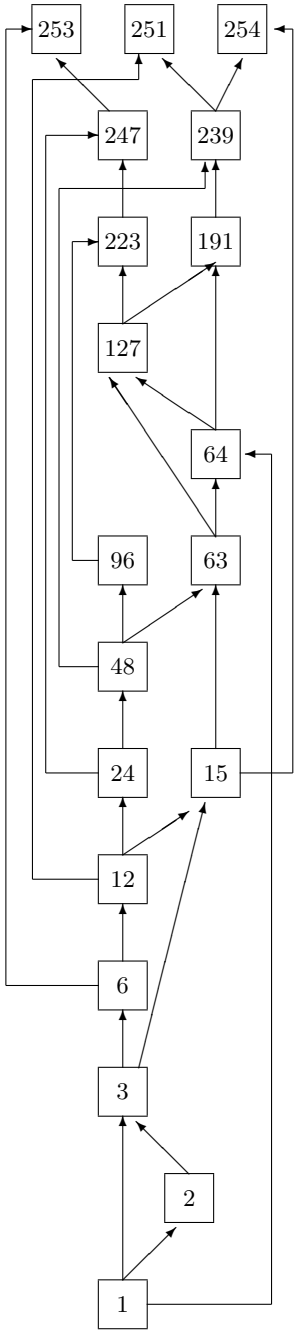

Fig. 3. Data flow graph of our addition chain

Practical Covertly Secure MPC for Dishonest Majority – or: Breaking the SPDZ Limits

Ivan Damgård[1], Marcel Keller[2], Enrique Larraia[2], Valerio Pastro[1], Peter Scholl[2], and Nigel P. Smart[2]

[1] Department of Computer Science, Aarhus University
[2] Department of Computer Science, University of Bristol

Abstract. SPDZ (pronounced "Speedz") is the nickname of the MPC protocol of Damgård et al. from Crypto 2012. SPDZ provided various efficiency innovations on both the theoretical and practical sides compared to previous work in the preprocessing model. In this paper we both resolve a number of open problems with SPDZ; and present several theoretical and practical improvements to the protocol.

In detail, we start by designing and implementing a covertly secure key generation protocol for obtaining a BGV public key and a shared associated secret key. In prior work this was assumed to be provided by a given setup functionality. Protocols for generating such shared BGV secret keys are likely to be of wider applicability than to the SPDZ protocol alone.

We then construct both a covertly and actively secure preprocessing phase, both of which compare favourably with previous work in terms of efficiency and provable security.

We also build a new online phase, which solves a major problem of the SPDZ protocol: namely prior to this work preprocessed data could be used for only one function evaluation and then had to be recomputed from scratch for the next evaluation, while our online phase can support reactive functionalities. This improvement comes mainly from the fact that our construction does not require players to reveal the MAC keys to check correctness of MAC'd values.

Since our focus is also on practical instantiations, our implementation offloads as much computation as possible into the preprocessing phase, thus resulting in a faster online phase. Moreover, a better analysis of the parameters of the underlying cryptoscheme and a more specific choice of the field where computation is performed allow us to obtain a better optimized implementation. Improvements are also due to the fact that our construction is in the random oracle model, and the practical implementation is multi-threaded.

This article is based on an earlier article: ESORICS 2013, pp 1–18, Springer LNCS 8134, 2013, http://dx.doi.org/10.1007/978-3-642-40203-6_1.

1 Introduction

For many decades multi-party computation (MPC) had been a predominantly theoretic endeavour in cryptography, but in recent years interest has arisen on the practical side. This has resulted in various implementation improvements and such protocols are becoming more applicable to practical situations. A key part in this transformation from theory to practice is in adapting theoretical protocols and applying implementation techniques so as to significantly improve performance, whilst not sacrificing the level of security required by real world applications. This paper follows this modern, more practical, trend.

Early applied work on MPC focused on the case of protocols secure against passive adversaries, both in the case of two-party protocols based on Yao circuits [19] and that of many-party protocols, based on secret sharing techniques [5, 10, 25]. Only in recent years work has shifted to achieve active security [17, 18, 24], which appears to come at vastly increased cost when dealing with more than two players. On the other hand, in the real applications active security may be more stringent than one would actually require. In [2, 3] Aumann and Lindell introduced the notion of covert security; in this security model an adversary who deviates from the protocol is detected with high (but not necessarily overwhelming) probability, say 90%, which still translates into an incentive on the adversary to behave in an honest manner. In contrast active security achieves the same effect, but the adversary can only succeed with cheating with negligible probability. There is a strong case to be made, see [2, 3], that covert security is a "good enough" security level for practical application; thus in this work we focus on covert security, but we also provide solutions with active security.

As our starting point we take the protocol of [14] (dubbed SPDZ, and pronounced Speedz). In [14] this protocol is secure against active static adversaries in the standard model, is actively secure, and tolerates corruption of $n - 1$ of the n parties. The SPDZ protocol follows the preprocessing model: in an offline phase some shared randomness is generated, but neither the function to be computed nor the inputs need be known; in an online phase the actual secure computation is performed. One of the main advantages of the SPDZ protocol is that the performance of the online phase scales linearly with the number of players, and the basic operations are almost as cheap as those used in the passively secure protocols based on Shamir secret sharing. Thus, it offers the possibility of being both more flexible and secure than Shamir based protocols, while still maintaining low computational cost.

In [12] the authors present an implementation report on an adaption of the SPDZ protocol in the random oracle model, and show performance figures for both the offline and online phases for both an actively secure variant and a covertly secure variant. The implementation is over a finite field of characteristic two, since the focus is on providing a benchmark for evaluation of the AES circuit (a common benchmark application in MPC [24, 11]).

Our Contributions: In this work we present a number of contributions which extend even further the ability the SPDZ protocol to deal with the type of application one is likely to see in practice. All our theorems are proved in the UC model, and in most cases, the protocols make use of some predefined ideal functionalities. We give protocols implementing most of these functionalities, the only exception being the functionality that provides access to a random oracle. This is implemented using a hash functions, and so the actual protocol is only secure in the Random Oracle Model. We back up these improvements with an implementation which we report on.

Our contributions come in two flavours. In the first flavour we present a number of improvements and extensions to the basic underlying SPDZ protocol. These protocol improvements are supported with associated security models and proofs. Our second flavour of improvements are at the implementation layer, and they bring in standard techniques from applied cryptography to bear onto MPC.

In more detail our protocol enhancements, in what are the descending order of importance, are as follows:

1. In the online phase of the original SPDZ protocol the parties are required to reveal their shares of a global MAC key in order to verify that the computation has been performed correctly. This is a major problem in practical applications since it means that secret-shared data we did not reveal cannot be re-used in later applications. Our protocol adopts a method to accomplish the same task, without needing to open the underlying MAC key. This means we can now go on computing on any secret-shared data we have, so we can support general reactive computation rather than just secure function evaluation. A further advantage of this technique is that some of the verification we need (the so-called "sacrificing" step) can be moved into the offline phase, providing additional performance improvements in the online phase.

2. In the original SPDZ protocol [12, 14] the authors assume a "magic" key generation phase for the production of the distributed Somewhat Homomorphic Encryption (SHE) scheme public/private keys required by the offline phase. The authors claim this can be accomplished using standard generic MPC techniques, which are of course expensive. In this work we present a key generation protocol for the BGV [6] SHE scheme, which is secure against covert adversaries. In addition we generate a "full" BGV key which supports the modulus switching and key switching used in [16]. This new sub-protocol may be of independent interest in other applications which require distributed decryption in an SHE/FHE scheme.
3. In [12] the modification to covert security was essentially ad-hoc, and resulted in a very weak form of covert security. In addition no security proofs or model were given to justify the claimed security. In this work we present a completely different approach to achieving covert security, we provide an extensive security model and provide full proofs for the modified offline phase (and the key generation protocol mentioned above).
4. We introduce a new approach to obtain full active security in the offline phase. In [14] active security was obtained via the use of specially designed ZKPoKs. In this work we present a different technique, based on a method used in [21]. This method has running time similar to the ZKPoK approach utilized in [14], but it allows us to give much stronger guarantees on the ciphertexts produced by corrupt players: the gap between the size of "noise" honest players put into ciphertexts and what we can force corrupt players to use was exponential in the security parameter in [14], and is essentially linear in our solution. This allows us to choose smaller parameters for the underlying cryptosystem and so makes other parts of the protocol more efficient.

It is important to understand that by combining these contributions in different ways, we can obtain two different general MPC protocols: First, since our new online phase still has full active security, it can be combined with our new approach to active security in the offline phase. This results in a protocol that is "syntactically similar" to the one from [14]: it has full active security assuming access to a functionality for key generation. However, it has enhanced functionality and performance, compared to [14], in that it can securely compute reactive functionalities. Second, we can combine our covertly secure protocols for key generation and the offline phase with the online phase to get a protocol that has covert security throughout and does not assume that key generation is given for free.

Our covert solutions all make use of the same technique to move from passive to covert security, while avoiding the computational cost of performing zero-knowledge proofs. In [12] covert security is obtained by only checking a fraction of the resulting proofs, which results in a weak notion of covert security (the probability of a cheater being detected cannot be made too large). In this work we adopt a different approach, akin to the cut-and-choose paradigm. We require parties to commit to random seeds for a number of runs of a given sub-protocol, then all the runs are executed in parallel, finally all bar one of the runs are "opened" by the players revealing their random seeds. If all opened runs are shown to have been performed correctly then the players assume that the single un-opened run is also correctly executed.

Note that since these checks take place in the offline phase where the inputs are not yet available, we obtain the strongest flavour of covert security defined in [2], where the adversary learns nothing new if he decides to try to cheat and is caught.

A pleasing side-effect of the replacement of zero-knowledge proofs with our custom mechanism to obtain covert security is that the offline phase can be run in much smaller "batches". In [12, 14] the need to amortize the cost of the expensive zero-knowledge proofs meant that the players on each iteration of the offline protocol executed a large computation, which produced a large number of multiplication triples [4] (in the millions). With our new technique we no longer need to amortize executions as much, and so short runs of the offline phase can be executed if so desired; producing only a few thousand triples per run.

Our second flavour of improvements at the implementation layer are more mundane; being mainly of an implementation nature.

1. We focus on the more practical application scenario of developing MPC where the base arithmetic domain is a finite field of characteristic $p > 2$. The reader should think $p \approx 2^{32}, 2^{64}, 2^{128}$ and the type of operations envisaged in [8, 9] etc. For such applications we can offload a lot of computation into the SPDZ offline phase, and we present the necessary modifications to do so.
2. Parameters for the underlying BGV scheme are chosen using the analysis used in [16] rather than the approach used in [14]. In addition we pick specific parameters which enable us to optimize for our application to SPDZ with the choices of p above.

3. We assume the random oracle model throughout, this allows us to simplify a number of the sub-procedures in [14]; especially, related to aspects of the protocol which require commitments.
4. The underlying modular arithmetic is implemented using Montgomery arithmetic [20], this is contrasted to earlier work which used standard libraries, such as NTL, to provide such operations.
5. The removal of the need to use libraries such as NTL means the entire protocol can be implemented in a multi-threaded manner; thus it can make use of the multiple cores on modern microprocessors.

This extended abstract presents the main ideas behind our improvements and details of our implementation. For a full description including details of the associated sub-procedures, security models and associated full security proofs please see the full version of this paper at [13].

2 SPDZ Overview

We now present the main components of the SPDZ protocol; in this section unless otherwise specified we are simply recapping on prior work. Throughout the paper we assume the computation to be performed by n players over a fixed finite field $\mathbb{F}_p$ of characteristic p. The high level idea of the online phase is to compute a function represented as a circuit, where privacy is obtained by additively secret sharing the inputs and outputs of each gate, and correctness is guaranteed by adding additive secret sharings of MACs on the inputs and outputs of each gate. In more detail, each player P_i has a uniform share $\alpha_i \in \mathbb{F}_p$ of a secret value $\alpha = \alpha_1 + \cdots + \alpha_n$, thought of as a fixed MAC key. We say that a data item $a \in \mathbb{F}_p$ is $\langle\cdot\rangle$-shared if P_i holds a tuple $(a_i, \gamma(a)_i)$, where a_i is an additive secret sharing of a, i.e. $a = a_1 + \cdots + a_n$, and $\gamma(a)_i$ is an additive secret sharing of $\gamma(a) := \alpha \cdot a$, i.e. $\gamma(a) = \gamma(a)_1 + \cdots + \gamma(a)_n$.

For the readers familiar with [14], this is a simpler MAC definition. In particular we have dropped δ_a from the MAC definition; this value was only used to add or subtract public data to or from shares. In our case δ_a becomes superfluous, since there is a straightforward way of computing a MAC of a public value a by defining $\gamma(a)_i \leftarrow a \cdot \alpha_i$.

During the protocol various values which are $\langle\cdot\rangle$-shared are "partially opened", i.e. the associated values a_i are revealed, but not the associated shares of the MAC. Note that linear operations (addition and scalar multiplication) can be performed on the $\langle\cdot\rangle$-sharings with no interaction required. Computing multiplications, however, is not straightforward, as we describe below.

The goal of the offline phase is to produce a set of "multiplication triples", which allow players to compute products. These are a list of sets of three $\langle\cdot\rangle$-sharings $\{\langle a\rangle, \langle b\rangle, \langle c\rangle\}$ such that $c = a \cdot b$. In this paper we extend the offline phase to also produce "square pairs" i.e. a list of pairs of $\langle\cdot\rangle$-sharings $\{\langle a\rangle, \langle b\rangle\}$ such that $b = a^2$, and "shared bits" i.e. a list of single shares $\langle a\rangle$ such that $a \in \{0, 1\}$.

In the online phase these lists are consumed as MPC operations are performed. In particular to multiply two $\langle\cdot\rangle$-sharings $\langle x\rangle$ and $\langle y\rangle$ we take a multiplication triple $\{\langle a\rangle, \langle b\rangle, \langle c\rangle\}$ and partially open $\langle x\rangle - \langle a\rangle$ to obtain ϵ and $\langle y\rangle - \langle b\rangle$ to obtain δ. The sharing of $z = x \cdot y$ is computed from $\langle z\rangle \leftarrow \langle c\rangle + \epsilon \cdot \langle b\rangle + \delta \cdot \langle a\rangle + \epsilon \cdot \delta$.

The reason for us introducing square pairs is that squaring a value can then be computed more efficiently as follows: To square the sharing $\langle x\rangle$ we take a square pair $\{\langle a\rangle, \langle b\rangle\}$ and partially open $\langle x\rangle - \langle a\rangle$ to obtain ϵ. We then compute the sharing of $z = x^2$ from $\langle z\rangle \leftarrow \langle b\rangle + 2 \cdot \epsilon \cdot \langle x\rangle - \epsilon^2$. Finally, the "shared bits" are useful in computing high level operation such as comparison, bit-decomposition, fixed and floating point operations as in [1, 8, 9].

The offline phase produces the triples in the following way. We make use of a Somewhat Homomorphic Encryption (SHE) scheme, which encrypts messages in $\mathbb{F}_p$, supports distributed decryption, and allows computation of circuits of multiplicative depth one on encrypted data. To generate a multiplication triple each player P_i generates encryptions of random values a_i and b_i (their shares of a and b). Using the multiplicative property of the SHE scheme an encryption of $c = (a_1 + \cdots + a_n) \cdot (b_1 + \cdots + b_n)$ is produced. The players then use the distributed decryption protocol to obtain sharings of c. The shares of the MACs on a, b and c needed to complete the $\langle\cdot\rangle$-sharing are produced in much the same manner. Similar operations are performed to produce square pairs and shared bits. Clearly the above (vague) outline needs to be fleshed out to ensure the required covert security level. Moreover, in practice we generate many triples/pairs/shared-bits at once using the SIMD nature of the BGV SHE scheme.

3 BGV

We now present an overview of the BGV scheme as required by our offline phase. This is only sketched, the reader is referred to [6, 15, 16] for more details; our goal is to present enough detail to explain the key generation protocol later.

3.1 Preliminaries

Underlying Algebra: We fix the ring $R_q = (\mathbb{Z}/q\mathbb{Z})[X]/\Phi_m(X)$ for some cyclotomic polynomial $\Phi_m(X)$, where m is an parameter to be determined later (see Appendix G). Note that q may not necessarily be prime. Let $R = \mathbb{Z}[X]/\Phi_m(X)$, and $\phi(m)$ denote the degree of R over $\mathbb{Z}$, i.e. Euler's ϕ function. The message space of our scheme will be R_p for a prime p of approximately 32, 64 or 128-bits in length, whilst ciphertexts will lie in either $R_{q_0}^2$ or $R_{q_1}^2$, for one of two moduli q_0 and q_1. We select $R = \mathbb{Z}[X]/(X^{m/2}+1)$ for m a power of two, and $p = 1 \pmod m$. By picking m and p this way we have that the message space R_p offers $m/2$-fold SIMD parallelism, i.e. $R_p \cong \mathbb{F}_p^{m/2}$. In addition this also implies that the ring constant c_m from [14, 16] is equal to one.

We wish to generate a public key for a leveled BGV scheme for which n players each hold a share, which is itself a "standard" BGV secret key. As we are working with circuits of multiplicative depth at most one we only need two levels in the moduli chain $q_0 = p_0$ and $q_1 = p_0 \cdot p_1$. The modulus p_1 will also play the role of P in [16] for the SwitchKey operation. The value p_1 must be chosen so that $p_1 \equiv 1 \pmod p$, with the value of p_0 set to ensure valid distributed decryption.

Random Values: Each player is assumed to have a secure entropy source. In practice we take this to be `/dev/urandom`, which is a non-blocking entropy source found on Unix like operating systems. This is not a "true" entropy source, being non-blocking, but provides a practical balance between entropy production and performance for our purposes. In what follows we model this source via a procedure $s \leftarrow \mathsf{Seed}()$, which generates a new seed from this source of entropy. Calling this function sets the players global variable cnt to zero. Then every time a player generates a new random value in a protocol this is constructed by calling $\mathsf{PRF}_s(\mathsf{cnt})$, for some pseudo-random function PRF, and then incrementing cnt. In practice we use AES under the key s with message cnt to implement PRF.

The point of this method for generating random values is that the said values can then be verified to have been generated honestly by revealing s in the future and recomputing all the randomness used by a player, and verifying his output is consistent with this value of s.

From the basic PRF we define the following "induced" pseudo-random number generators, which generate elements according to the following distributions but seeded by the seed s:

- $\mathcal{HWT}_s(h, n)$: This generates a vector of length n with elements chosen at random from $\{-1, 0, 1\}$ subject to the condition that the number of non-zero elements is equal to h.
- $\mathcal{ZO}_s(0.5, n)$: This generates a vector of length n with elements chosen from $\{-1, 0, 1\}$ such that the probability of coefficient is $p_{-1} = 1/4$, $p_0 = 1/2$ and $p_1 = 1/4$.
- $\mathcal{DG}_s(\sigma^2, n)$: This generates a vector of length n with elements chosen according to the discrete Gaussian distribution with variance σ^2.
- $\mathcal{RC}_s(0.5, \sigma^2, n)$: This generates a triple of elements (v, e_0, e_1) where v is sampled from $\mathcal{ZO}_s(0.5, n)$ and e_0 and e_1 are sampled from $\mathcal{DG}_s(\sigma^2, n)$.
- $\mathcal{U}_s(q, n)$: This generates a vector of length n with elements generated uniformly modulo q.

If any random values are used which **do not** depend on a seed then these should be assumed to be drawn using a secure entropy source (again in practice assumed to be `/dev/urandom`). If we pull from one of the above distributions where we do not care about the specific seed being used then we will drop the subscript s from the notation.

Broadcast: When broadcasting data we assume two different models. In the online phase during partial opening we utilize the method described in [14]; in that players send their data to a nominated player who then broadcasts the reconstructed value back to the remaining players. For other applications of broadcast we assume each party broadcasts their values to all other parties directly. In all instances players maintain a running hash of all values sent and received in a broadcast (with a suitable modification for the variant used for partial opening). At the end of a protocol run these

running hashes are compared in a pair-wise fashion. This final comparison ensures that in the case of at least two honest parties the adversary must have been consistent in what was sent to the honest parties.

Commitments: In Figure 2 we present an ideal functionality $\mathcal{F}_{\text{COMMIT}}$ for commitment which will be used in all of our protocols. Our protocols will be UC secure, this is possible despite the fact that we allow dishonest majority because we assume a random oracle is available; in particular we model a hash function $\mathcal{H}_1$ as a random oracle and define a commitment scheme to implement the functionality $\mathcal{F}_{\text{COMMIT}}$ as follows: The commit function $\mathsf{Commit}(m)$ generates a random value r and computes $c \leftarrow \mathcal{H}_1(m\|r)$. It returns the pair (c, o) where o is the opening information $m\|r$. When the commitment c is opened the committer outputs the value o and the receiver runs $\mathsf{Open}(c, o)$ which checks whether $c = \mathcal{H}_1(o)$ and if the check passes it returns m. See Appendix A for details.

3.2 Key Generation

The key generation algorithm generates a public/private key pair such that the public key is given by $\mathfrak{pk} = (a, b)$, where a is generated from $\mathcal{U}(q_1, \phi(m))$ (i.e. a is uniform in R_{q_1}), and $b = a \cdot \mathfrak{s} + p \cdot \epsilon$ where ϵ is a "small" error term, and $\mathfrak{s}$ is the secret key such that $\mathfrak{s} = \mathfrak{s}_1 + \cdots + \mathfrak{s}_n$, where player P_i holds the share $\mathfrak{s}_i$. Recall since m is a power of 2 we have $\phi(m) = m/2$.

The public key is also augmented to an extended public key $\mathfrak{epk}$ by addition of a "quasi-encryption" of the message $-p_1 \cdot \mathfrak{s}^2$, i.e. $\mathfrak{epk}$ contains a pair $\mathfrak{enc} = (b_{\mathfrak{s},\mathfrak{s}^2}, a_{\mathfrak{s},\mathfrak{s}^2})$ such that $b_{\mathfrak{s},\mathfrak{s}^2} = a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} + p \cdot \epsilon_{\mathfrak{s},\mathfrak{s}^2} - p_1 \cdot \mathfrak{s}^2$, where $a_{\mathfrak{s},\mathfrak{s}^2} \leftarrow \mathcal{U}(q_1, \phi(m))$ and $\epsilon_{\mathfrak{s},\mathfrak{s}^2}$ is a "small" error term. The precise distributions of all these values will be determined when we discuss the exact key generation protocol we use.

3.3 Encryption and Decryption

$\mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m})$: To encrypt an element $m \in R_p$, using the modulus q_1, we choose one "small polynomial" (with $0, \pm 1$ coefficients) and two Gaussian polynomials (with variance σ^2), via $(v, e_0, e_1) \leftarrow \mathcal{RC}_s(0.5, \sigma^2, \phi(m))$. Then we set $c_0 = b \cdot v + p \cdot e_0 + m,\ c_1 = a \cdot v + p \cdot e_1$, and set the initial ciphertext as $\mathfrak{c}' = (c_0, c_1, 1)$.

$\mathsf{SwitchModulus}((c_0, c_1), \ell)$: The operation $\mathsf{SwitchModulus}(\mathfrak{c})$ takes the ciphertext $\mathfrak{c} = ((c_0, c_1), \ell)$ defined modulo q_ℓ and produces a ciphertext $\mathfrak{c}' = ((c_0', c_1'), \ell - 1)$ defined modulo $q_{\ell-1}$, such that $[c_0 - \mathfrak{s} \cdot c_1]_{q_\ell} \equiv [c_0' - \mathfrak{s} \cdot c_1']_{q_{\ell-1}} \pmod{p}$. This is done by setting $c_i' = \mathsf{Scale}(c_i, q_\ell, q_{\ell-1})$ where Scale is the function defined in [16]; note we need the more complex function of Appendix E of the full version of [16] if working in dCRT representation as we need to fix the scaling modulo p as opposed to modulo two which was done in the main body of [16]. As we are only working with two levels this function can only be called when $\ell = 1$.

$\mathsf{Dec}_{\mathfrak{s}}(\mathfrak{c})$: Note, that this operation is never actually performed, since no-one knows the shared secret key $\mathfrak{s}$, but presenting it will be instructive: Decryption of a ciphertext (c_0, c_1, ℓ) at level ℓ is performed by setting $m' = [c_0 - \mathfrak{s} \cdot c_1]_{q_\ell}$, then converting m' to coefficient representation and outputting $m' \bmod p$.

$\mathsf{DistDec}_{\mathfrak{s}_i}(\mathfrak{c})$: We actually decrypt using a simplification of the distributed decryption procedure described in [14], since our final ciphertexts consist of only two elements as opposed to three in [14]. For input ciphertext (c_0, c_1, ℓ), player P_1 computes $\mathbf{v}_1 = c_0 - \mathfrak{s}_i \cdot c_1$ and each other player P_i computes $\mathbf{v}_i = -\mathfrak{s}_i \cdot c_1$. Each party P_i then sets $\mathbf{t}_i = \mathbf{v}_i + p \cdot \mathbf{r}_i$ for some random element $\mathbf{r}_i \in R$ with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$, for some statistical security parameter sec, and the values $\mathbf{t}_i$ are broadcast; the precise value B being determined in Appendix G. Then the message is recovered as $\mathbf{t}_1 + \cdots + \mathbf{t}_n \pmod{p}$.

3.4 Operations on Encrypted Data

Homomorphic addition follows trivially from the methods of [6, 16]. So the main remaining task is to deal with multiplication. We first define a $\mathsf{SwitchKey}$ operation.

$\underline{\mathsf{SwitchKey}(d_0, d_1, d_2)}$: This procedure takes as input an extended ciphertext $\mathfrak{c} = (d_0, d_1, d_2)$ defined modulo q_1; this is a ciphertext which is decrypted via the equation

$$[d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}^2 \cdot d_2]_{q_1}.$$

The SwitchKey operation also takes the key-switching data $\mathfrak{enc} = (b_{\mathfrak{s},\mathfrak{s}^2}, a_{\mathfrak{s},\mathfrak{s}^2})$ above and produces a standard two element ciphertext which encrypts the same message but modulo q_0.

- $c_0' \leftarrow p_1 \cdot d_0 + b_{\mathfrak{s},\mathfrak{s}^2} \cdot d_2 \pmod{q_1}$, $\quad c_1' \leftarrow p_1 \cdot d_1 + a_{\mathfrak{s},\mathfrak{s}^2} \cdot d_2 \pmod{q_1}$.
- $c_0'' \leftarrow \mathsf{Scale}(c_0', q_1, q_0)$, $\quad c_1'' \leftarrow \mathsf{Scale}(c_1', q_1, q_0)$.
- Output $((c_0'', c_1''), 0)$.

Notice we have the following equality modulo q_1:

$$\begin{aligned} c_0' - \mathfrak{s} \cdot c_1' &= (p_1 \cdot d_0) + d_2 \cdot b_{\mathfrak{s},\mathfrak{s}^2} - \mathfrak{s} \cdot \big((p \cdot d_1) - d_2 \cdot a_{\mathfrak{s},\mathfrak{s}^2}\big) \\ &= p_1 \cdot (d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}^2 d_2) - p \cdot d_2 \cdot \epsilon_{\mathfrak{s},\mathfrak{s}^2}, \end{aligned}$$

The requirement on $p_1 \equiv 1 \pmod{p}$ is from the above equation as we want this to produce the same value as $d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}^2 d_2 \mod q_1$ on reduction modulo p.

$\underline{\mathsf{Mult}(\mathfrak{c}, \mathfrak{c}')}$: We only need to execute multiplication on two ciphertexts at level one, thus $\mathfrak{c} = ((c_0, c_1), 1)$ and $\mathfrak{c}' = ((c_0', c_1'), 1)$. The output will be a ciphertext $\mathfrak{c}''$ at level zero, obtained via the following steps:

- $\mathfrak{c} \leftarrow \mathsf{SwitchModulus}(\mathfrak{c})$, $\quad \mathfrak{c}' \leftarrow \mathsf{SwitchModulus}(\mathfrak{c}')$.
- $(d_0, d_1, d_2) \leftarrow (c_0 \cdot c_0', c_1 \cdot c_0' + c_0 \cdot c_1', -c_1 \cdot c_1')$.
- $\mathfrak{c}'' \leftarrow \mathsf{SwitchKey}(d_0, d_1, d_2)$.

4 Protocols Associated to the SHE Scheme

In this section we present two sub-protocols associated with the SHE scheme; namely our distributed key generation and a protocol for proving that a committed ciphertext is well formed.

4.1 Distributed Key Generation Protocol For BGV

To make the paper easier to follow we present the precise protocols, ideal functionalities, simulators and security proofs in Appendix B. Here we present a high level overview.

As remarked in the introduction, the authors of [14] assumed a "magic" set up which produces not only a distributed sharing of the main BGV secret key, but also a distributed sharing of the square of the secret key. That was assumed to be done via some other unspecified MPC protocol. The effect of requiring a sharing of the square of the secret key was that they did not need to perform KeySwitching, but ciphertexts were 50% bigger than one would otherwise expect. Here we take a very different approach: we augment the public key with the keyswitching data from [16] and provide an explicit covertly secure key generation protocol.

Our protocol will be covertly secure in the sense that the probability that an adversary can deviate without being detected will be bounded by $1/c$, for a positive integer c. Our basic idea behind achieving covert security is as follows: Each player runs c instances of the basic protocol, each with different random seeds, then at the end of the main protocol all bar a random one basic protocol runs are opened, along with the respective random seeds. All parties then check that the opened runs were performed honestly and, if any party finds an inconsistency, the protocol aborts. If no problem is detected, the parties assume that the single unopened run is correct. Thus intuitively the adversary can cheat with probability at most $1/c$.

We start by discussing the generation of the main public key $\mathfrak{pk}_j$ in execution j where $j \in \{1, \ldots, c\}$. To start with the players generate a uniformly random value $a_j \in R_{q_1}$. They then each execute the standard BGV key generation

procedure, except that this is done with respect to the global element a_j. Player i chooses a low-weight secret key and then generates an LWE instance relative to that secret key. Following [16], we choose

$$\mathfrak{s}_{i,j} \leftarrow \mathcal{HWT}_s(h, \phi(m)) \text{ and } \epsilon_{i,j} \leftarrow \mathcal{DG}_s(\sigma^2, \phi(m)).$$

Then the player sets the secret key as $\mathfrak{s}_{i,j}$ and their "local" public key as $(a_j, b_{i,j})$ where $b_{i,j} = [a_j \cdot \mathfrak{s}_{i,j} + p \cdot \epsilon_{i,j}]_{q_1}$.

Note, by a hybrid argument, obtaining n ring-LWE instances for n different secret keys but the same value of a_j is secure assuming obtaining one ring-LWE instance is secure. In the LWE literature this is called "amortization". Also note in what follows that a key modulo q_1 can be also treated as a key modulo q_0 since q_0 divides q_1 and $\mathfrak{s}_{i,j}$ has coefficients in $\{-1, 0, 1\}$.

The global public and private key are then set to be $\mathfrak{pk}_j = (a_j, b_j)$ and $\mathfrak{s}_j = \mathfrak{s}_{1,j} + \cdots + \mathfrak{s}_{n,j}$, where $b_j = [b_{1,j} + \cdots + b_{n,j}]_{q_1}$. This is essentially another BGV key pair, since if we set $\epsilon_j = \epsilon_{1,j} + \cdots + \epsilon_{n,j}$ then we have

$$b_j = \sum_{i=1}^{n} (a_j \cdot \mathfrak{s}_{i,j} + p \cdot \epsilon_{i,j}) = a_j \cdot \mathfrak{s}_j + p \cdot \epsilon_j,$$

but generated with different distributions for $\mathfrak{s}_j$ and ϵ_j compared to the individual key pairs above.

We next augment the above basic key generation to enable the construction of the KeySwitching data. Given a public key $\mathfrak{pk}_j$ and a share of the secret key $\mathfrak{s}_{i,j}$ our method for producing the extended public key is to produce in turn (see Figure 3 for the details on how we create these elements in our protocol).

- $\mathfrak{enc}'_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}_j}(-p_1 \cdot \mathfrak{s}_{i,j})$
- $\mathfrak{enc}'_j \leftarrow \mathfrak{enc}'_{1,j} + \cdots + \mathfrak{enc}'_{n,j}$.
- $\mathfrak{zero}_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}_j}(\mathbf{0})$
- $\mathfrak{enc}_{i,j} \leftarrow (\mathfrak{s}_{i,j} \cdot \mathfrak{enc}'_j) + \mathfrak{zero}_{i,j} \in R^2_{q_1}$.
- $\mathfrak{enc}_j \leftarrow \mathfrak{enc}_{1,j} + \cdots + \mathfrak{enc}_{n,j}$.
- $\mathfrak{epk}_j \leftarrow (\mathfrak{pk}_j, \mathfrak{enc}_j)$.

Note, that $\mathfrak{enc}'_{i,j}$ is not a valid encryption of $-p_1 \cdot \mathfrak{s}_{i,j}$, since $-p_1 \cdot \mathfrak{s}_{i,j}$ does not lie in the message space of the encryption scheme. However, because of the dependence on the secret key shares here, we need to assume a form of circular security; the precise assumption needed is stated in Appendix B. The encryption of zero, $\mathfrak{zero}_{i,j}$, is added on by each player to re-randomize the ciphertext, preventing an adversary from recovering $\mathfrak{s}_{i,j}$ from $\mathfrak{enc}_{i,j}/\mathfrak{enc}'_j$. We call the resulting $\mathfrak{epk}_j$ the *extended public key*. In [16] the keyswitching data $\mathfrak{enc}_j$ is computed directly from $\mathfrak{s}_j^2$; however, we need to use the above round-about way since $\mathfrak{s}_j^2$ is not available to the parties.

Finally we open all bar one of the c executions and check they have been executed correctly. If all checks pass then the final extended public key $\mathfrak{epk}$ is output and the players keep hold of their associated secret key share $\mathfrak{s}_i$. See Figure 3 for full details of the protocol.

Theorem 1. *In the $\mathcal{F}_{\text{COMMIT}}$-hybrid model, the protocol Π_{KEYGEN} implements $\mathcal{F}_{\text{KEYGEN}}$ with computational security against any static adversary corrupting at most $n-1$ parties.*

Recall that $\mathcal{F}_{\text{COMMIT}}$ is a standard functionality for commitment. $\mathcal{F}_{\text{KEYGEN}}$ simply generates a key pair with a distribution matching what we sketched above, and then sends the values $a_i, b_i, \mathfrak{enc}'_i, \mathfrak{enc}_i$ for every i to all parties and shares of the secret key to the honest players. Like most functionalities in the following, it allows the adversary to try to cheat and will allow this with a certain probability $1/c$. This is how we model covert security.

The BGV cryptosystem resulting from $\mathcal{F}_{\text{KEYGEN}}$ is proven semantically secure by the following theorem.

Theorem 2. *If the functionality $\mathcal{F}_{\text{KEYGEN}}$ is used to produce a public key $\mathfrak{epk}$ and secret keys $\mathfrak{s}_i$ for $i = 0, \ldots, n-1$ then the resulting cryptosystem is semantically secure based on the hardness of $\mathsf{RLWE}_{q_1,\sigma^2,h}$ and the circular security assumption in Appendix B.*

4.2 EncCommit

We use a sub-protocol $\Pi_{\text{EncCommit}}$ to replace the Π_{ZKPoPK} protocol from [14]. In this section we consider a covertly secure variant rather than active security; this means that players controlled by a malicious adversary succeed in deviating from the protocol with a probability bounded by $1/c$. In our experiments we pick $c = 5$, 10 and 20. In Appendix F we present an actively secure variant of this protocol.

Our new sub-protocol assumes that players have agreed on the key material for the encryption scheme, i.e. $\Pi_{\text{EncCommit}}$ runs in the $\mathcal{F}_{\text{KeyGen}}$-hybrid model. The protocol ensures that a party outputs a validly created ciphertext containing an encryption of some pseudo-random message m, where the message m is drawn from a distribution satisfying condition cond. This is done by committing to seeds and using the cut-and-choose technique, similarly to the key generation protocol. The condition cond in our application could either be uniformly pseudo-randomly generated from R_p, or uniformly pseudo-randomly generated from $\mathbb{F}_p$ (i.e. a "diagonal" element in the SIMD representation).

The protocol $\Pi_{\text{EncCommit}}$ and ideal functionality it implements are presented in Appendix C, along with the proof of the following theorem.

Theorem 3. *In the $(\mathcal{F}_{\text{Commit}}, \mathcal{F}_{\text{KeyGen}})$-hybrid model, the protocol $\Pi_{\text{EncCommit}}$ implements $\mathcal{F}_{\text{SHE}}$ with computational security against any static adversary corrupting at most $n - 1$ parties.*

$\mathcal{F}_{\text{SHE}}$ offers the same functionality as $\mathcal{F}_{\text{KeyGen}}$ but can in addition generate correctly formed ciphertexts where the plaintext satisfies a condition cond as explained above, and where the plaintext is known to a particular player (even if he is corrupt). Of course, if we use the actively secure version of $\Pi_{\text{EncCommit}}$ from Appendix F, we would get a version of $\mathcal{F}_{\text{SHE}}$ where the adversary is not allowed to attempt cheating.

5 The Offline Phase

The offline phase produces pre-processed data for the online phase (where the secure computation is performed). To ensure security against active adversaries the MAC values of any partially opened value need to be verified. We suggest a new method for this that overcomes some limitations of the corresponding method from [14]. Since it will be used both in the offline and the online phase, we explain it here, before discussing the offline phase.

5.1 MAC Checking

We assume some value a has been $\langle\cdot\rangle$-shared and partially opened, which means that players have revealed shares of the a but not of the associated MAC value γ, this is still additively shared. Since there is no guarantee that the a are correct, we need to check it holds that $\gamma = \alpha a$ where α is the global MAC key that is also additively shared. In [14], this was done by having players commit to the shares of the MAC. then open α and check everything in the clear. But this means that other shared values become useless because the MAC key is now public, and the adversary could manipulate them as he desires.

So we want to avoid opening α, and observe that since a is public, the value $\gamma - \alpha a$ is a linear function of shared values γ, α, so players can compute shares in this value locally and we can then check if it is 0 without revealing information on α. As in [14], we can optimize the cost of this by checking many MACs in one go: we take a random linear combination of a and γ-values and check only the results of this. The full protocol is given in Figure 10; it is not intended to implement any functionality – it is just a procedure that can be called in both the offline and online phases. MACCheck has the following important properties.

Lemma 1. *The protocol* MACCheck *is* correct*, i.e. it accepts if all the values a_j and the corresponding MACs are correctly computed. Moreover, it is* sound*, i.e. it rejects except with probability $2/p$ in case at least one value or MAC is not correctly computed.*

The proof of Lemma 1 is given in Appendix D.3.

5.2 Offline Protocol

The offline phase itself runs two distinct sub-phases, each of which we now describe. To start with we assume a BGV key has been distributed according to the key generation procedure described earlier, as well as the shares of a secret MAC key and an encryption $\mathfrak{c}_\alpha$ of the MAC key as above. We assume that the output of the offline phase will be a total of at least n_I input tuples, n_m multiplication triples, n_s squaring tuples and n_b shared bits.

In the first sub-phase, which we call the tuple-production sub-phase, we over-produce the various multiplication and squaring tuples, plus the shared bits. These are then "sacrificed" in the tuple-checking phase so as to create at least n_m multiplication triples, n_s squaring tuples and n_b shared bits. In particular in the tuple-production phase we produce (at least) $2 \cdot n_m$ multiplication tuples, $2 \cdot n_s + n_b$ squaring tuples, and n_b shared bits. Tuple-production is performed by following the protocol in Figure 13 and Figure 14. The tuple production protocol can be run repeatedly, alongside the tuple-checking sub-phase and the online phase.

The second sub-phase of the offline phase is to check whether the resulting material from the prior phase has been produced correctly. This check is needed, because the distributed decryption procedure needed to produce the tuples and the MACs could allow the adversary to induce errors. We solve this problem via a sacrificing technique, as in [14], however, we also need to adapt it to the case of squaring tuples and bit-sharings. Moreover, this sacrificing is performed in the offline phase as opposed to the online phase (as in [14]); and the resulting partially opened values are checked in the offline phase (again as opposed to the online phase). This is made possible by our protocol MACCheck which allows to verify the MACs are correct without revealing the MAC key α. The tuple-checking protocol is presented in Figure 15.

We show that the resulting protocol Π_{PREP}, given in Figure 12, securely implements the functionality $\mathcal{F}_{\text{PREP}}$, which models the offline phase. The functionality $\mathcal{F}_{\text{PREP}}$ outputs some desired number of multiplication triples, squaring tuples and shared bits. In Appendix D we present a proof of the following theorem.

Theorem 4. *In the $(\mathcal{F}_{\text{SHE}}, \mathcal{F}_{\text{COMMIT}})$-hybrid model, the protocol Π_{PREP} implements $\mathcal{F}_{\text{PREP}}$ with computational security against any static adversary corrupting at most $n-1$ parties if p is exponential in the security parameter.*

The security flavour of Π_{PREP} follows the security of EncCommit, i.e. if one uses the covert (resp. active) version of EncCommit, one gets covert (resp. active) security for Π_{PREP}.

6 Online Phase

We design a protocol Π_{ONLINE} which performs the secure computation of the desired function, decomposed as a circuit over $\mathbb{F}_p$. Our online protocol makes use of the preprocessed data coming from $\mathcal{F}_{\text{PREP}}$ in order to input, add, multiply or square values. Our protocol is similar to the one described in [14]; however, it brings a series of improvements, in the sense that we could push the "sacrificing" to the preprocessing phase, we have specialised procedure for squaring etc, and we make use of a different MAC-checking method in the output phase. Our method for checking the MACs is simply the MACCheck protocol on all partially opened values; note that such a method has a lower soundness error than the method proposed in [14], since the linear combination of partially opened values is truly random in our case, while it has lower entropy in [14].

The following theorem, whose proof is given in Appendix E, shows that the protocol Π_{ONLINE}, given in Figure 20, securely implements the functionality $\mathcal{F}_{\text{ONLINE}}$, which models the online phase.

Theorem 5. *In the $\mathcal{F}_{\text{PREP}}$-hybrid model, the protocol Π_{ONLINE} implements $\mathcal{F}_{\text{ONLINE}}$ with computational security against any static adversary corrupting at most $n-1$ parties if p is exponential in the security parameter.*

The astute reader will be wondering where our shared bits produced in the offline phase are used. These will be used in "higher level" versions of the online phase (i.e. versions which do not just evaluate an arithmetic circuit) which implement the types of operations presented in [8, 9].

7 Experimental Results

7.1 KeyGen and Offline Protocols

To present performance numbers for our key generation and new variant of the offline phase for SPDZ we first need to define secure parameter sizes for the underlying BGV scheme (and in particular how it is used in our protocols). This is done in Appendix G, by utilizing the method of Appendices A.4, A.5 and B of [16], for various choices of n (the number of players) and p (the field size).

We then implemented the preceding protocols in C++ on top of the MPIR library for multi-precision arithmetic. Modular arithmetic was implemented with bespoke code using Montgomery arithmetic [20] and calls to the underlying `mpn_` functions in MPIR. The offline phase was implemented in a multi-threaded manner, with four cores producing initial multiplication triples, square pairs, shared bits and input preparation mask values. Then two cores performed the sacrificing for the multiplication triples, square pairs and shared bits.

In Table 1 we present execution times (in wall time measured in seconds) for key generation and for an offline phase which produces 100000 each of the multiplication tuples, square pairs, shared bits and 1000 input sharings. We also present the average time to produce a multiplication triple for an offline phase running on one core and producing 100000 multiplication triples only. The run-times are given for various values of n, p and c, and all timings were obtained on 2.80 GHz Intel Core i7 machines with 4 GB RAM, with machines running on a local network.

			Run Times		Time per
n	$p \approx$	c	KeyGen	Offline	Triple (sec)
2	2^{32}	5	2.4	156	0.00140
2	2^{32}	10	5.1	277	0.00256
2	2^{32}	20	10.4	512	0.00483
2	2^{64}	5	5.9	202	0.00194
2	2^{64}	10	12.5	377	0.00333
2	2^{64}	20	25.6	682	0.00634
2	2^{128}	5	16.2	307	0.00271
2	2^{128}	10	33.6	561	0.00489
2	2^{128}	20	74.5	1114	0.00937

			Run Times		Time per
n	$p \approx$	c	KeyGen	Offline	Triple(sec)
3	2^{32}	5	3.0	292	0.00204
3	2^{32}	10	6.4	413	0.00380
3	2^{32}	20	13.3	790	0.00731
3	2^{64}	5	7.7	292	0.00267
3	2^{64}	10	16.3	568	0.00497
3	2^{64}	20	33.7	1108	0.01004
3	2^{128}	5	21.0	462	0.00402
3	2^{128}	10	44.4	889	0.00759
3	2^{128}	20	99.4	2030	0.01487

Table 1. Execution Times For Key Gen and Offline Phase (Covert Security)

We compare the results to that obtained in [12], since no other protocol can provide malicious/covert security for $t < n$ corrupted parties. In the case of covert security the authors of [12] report figures of 0.002 seconds per (un-checked) 64-bit multiplication triple for both two and three players; however the probability of cheating being detected was lower bounded by $1/2$ for two players, and $1/4$ for three players; as opposed to our probabilities of $4/5$, $9/10$ and $19/20$. Since the triples in [12] were unchecked we need to scale their run-times by a factor of two; to obtain 0.004 seconds per multiplication triple. Thus for covert security we see that our protocol for checked tuples are superior both in terms error probabilities, for a comparable run-time.

When using our active security variant we aimed for a cheating probability of 2^{-40}; so as to be able to compare with prior run times obtained in [12], which used the method from [14]. Again we performed two experiments one where four cores produced 100000 multiplication triples, squaring pairs and shared bits, plus 1000 input sharings; and one experiment where one core produced just 100000 multiplication triples (so as to produce the average cost for a triple). The results are in Table 2.

By way of comparison for a prime of 64 bits the authors of [12] report on an implementation which takes 0.006 seconds to produce an (un-checked) multiplication triple for the case of two parties and equivalent active security; and 0.008 per second for the case of three parties and active security. As we produce checked triples, the cost per triple for the results in [12] need to be (at least) doubled; to produce a total of 0.012 and 0.016 seconds respectively.

Thus, in this test, our new active protocol has running time about twice that of the previous active protocol from [14] based on ZKPoKs. From the analysis of the protocols, we do expect that the new method will be faster, but only if we produce the output in large enough batches. Due to memory constraints we were so far unable to do this, but we can extrapolate from these results: In the test we generated 12 ciphertexts in one go, and if we were able to increase

	$n = 2$		$n = 3$	
$p \approx$	Offline	Time per Triple	Offline	Time per Triple
2^{32}	2366	0.01955	3668	0.02868
2^{64}	3751	0.02749	5495	0.04107
2^{128}	6302	0.04252	10063	0.06317

Table 2. Execution Times For Offline Phase (Active Security)

this by a factor of about 10, then we would get results better than those of [14, 12], all other things being equal. More information can be found in Appendix F.

7.2 Online

For the new online phase we have developed a purpose-built bytecode interpreter, which reads and executes pre-generated sequences of instructions in a multi-threaded manner. Our runtime supports parallelism on two different levels: independent rounds of communication can be merged together to reduce network overhead, and multiple threads can be executed at once to allow for optimal usage of modern multi-core processors.

Each bytecode instruction is either some local computation (e.g. addition of secret shared values) or an 'open' instruction, which initiates the protocol to reveal a shared value. The data from independent open instructions can be merged together to save on communication costs. Each player may run up to four different bytecode files in parallel in distinct threads, with each such thread having access to some shared memory resource. The advantage of this approach is that bytecode files can be pre-compiled and optimized, and then quickly loaded at runtime – the online phase runtime is itself oblivious to the nature of the programs being run.

In Table 3 we present timings (again in elapsed wall time for a player) for multiplying two secret shared values. Results are given for three different varieties of multiplication, reflecting the possibilities available: purely sequential multiplications; parallel multiplications with communication merged into one round (50 per round); and parallel multiplications running in 4 independent threads (50 per round, per thread). The experiments were carried out on the same machines as the offline phase, running over a local network with a ping of around 0.27ms. For comparison, the original implementation of the online phase in [14] gave an amortized time of 20000 multiplications per second over a 64-bit prime field, with three players.

		Multiplications/sec		
		Sequential	50 in Parallel	
n	$p \approx$	Single Thread	Single Thread	Four Threads
2	2^{32}	7500	134000	398000
2	2^{64}	7500	130000	395000
2	2^{128}	7500	120000	358000
3	2^{32}	4700	100000	292000
3	2^{64}	4700	98000	287000
3	2^{128}	4600	90000	260000

Table 3. Online Times

8 Acknowledgements

The first and fourth author acknowledge partial support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, and from the CFEM research center (supported by the Danish Strategic Research Council). The second, third, fifth and sixth authors were supported by EPSRC via grant COED–EP/I03126X. The sixth author was also supported by the European Commission via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO,

the Defense Advanced Research Projects Agency and the Air Force Research Laboratory under agreement number FA8750-11-2-0079[3], and by a Royal Society Wolfson Merit Award.

References

1. M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele. Secure computation on floating point numbers. In *Network and Distributed System Security Symposium – NDSS 2013*. Internet Society, 2013.
2. Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *Theory of Cryptography – TCC 2007*, volume 4392 of *LNCS*, pages 137–156. Springer, 2007.
3. Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, 23(2):281–343, 2010.
4. D. Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, volume 576 of *LNCS*, pages 420–432. Springer, 1991.
5. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security – ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, 2008.
6. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, pages 309–325. ACM, 2012.
7. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from Ring-LWE and security for key dependent messages. In *CRYPTO*, volume 6841 of *LNCS*, pages 505–524. Springer, 2011.
8. O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In *Financial Cryptography – FC 2010*, volume 6052 of *LNCS*, pages 35–50. Springer, 2010.
9. I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography – TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, 2006.
10. I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Public Key Cryptography – PKC 2009*, volume 5443 of *LNCS*, pages 160–179. Springer, 2009.
11. I. Damgård and M. Keller. Secure multiparty AES. In *Financial Cryptography*, volume 6052 of *LNCS*, pages 367–374. Springer, 2010.
12. I. Damgård, M. Keller, E. Larraia, C. Miles, and N. P. Smart. Implementing aes via an actively/covertly secure dishonest-majority mpc protocol. In *SCN*, volume 7485 of *LNCS*, pages 241–263. Springer, 2012.
13. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits, 2012.
14. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, 2012.
15. C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 465–482. Springer, 2012.
16. C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, 2012.
17. B. Kreuter, A. Shelat, and C.-H. Shen. Towards billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium – 2012*, pages 285–300, 2012.
18. Y. Lindell, B. Pinkas, and N. P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *Security and Cryptography for Networks – SCN 2008*, volume 5229 of *LNCS*, pages 2–20. Springer, 2008.
19. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - Secure two-party computation system. In *USENIX Security Symposium – 2004*, pages 287–302, 2004.
20. P. L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44:519–521, 1985.
21. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, 2012.
22. J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. In *TCC*, volume 5444 of *LNCS*, pages 368–386. Springer, 2009.
23. C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. *Advances in Cryptology–CRYPTO 2008*, pages 554–571, 2008.

[3] The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, AFRL, or the U.S. Government.

24. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.
25. SIMAP Project. SIMAP: Secure information management and processing. `http://alexandra.dk/uk/Projects/Pages/SIMAP.aspx`.

A Commitments in the Random Oracle Model

A.1 Protocol and Functionality

The Protocol Π_{COMMIT}

Commit:

1. In order to commit to v, P_i sets $o \leftarrow v||r$, where r is chosen uniformly in a determined domain, and queries the Random Oracle $\mathcal{H}$ to get $c \leftarrow \mathcal{H}(o)$.
2. P_i then broadcasts (c, i, τ_v), where τ_v represents a handle for the commitment.

Open:

1. In order to open a commitment (c, i, τ_v), where $c = \mathcal{H}(v||r)$, player P_i broadcasts $(o = v||r, i, \tau_v)$.
2. All players call $\mathcal{H}$ on o and check whether $\mathcal{H}(o) = c$. Players accept if and only if this check passes.

Fig. 1. The Protocol for Commitments.

The Ideal Functionality $\mathcal{F}_{\text{COMMIT}}$

Commit: On input $(\mathsf{Commit}, v, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), where v is either in a specific domain or $\perp$, it stores (v, i, τ_v) on a list and outputs (i, τ_v) to all players and adversary.

Open: On input $(\mathsf{Open}, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), the ideal functionality outputs (v, i, τ_v) to all players and adversary. If $(\mathsf{NoOpen}, i, \tau_v)$ is given by the adversary, and P_i is corrupt, the functionality outputs $(\perp, i, \tau_v)$ to all players.

Fig. 2. The Ideal Functionality for Commitments

A.2 UC Security

Lemma 2. *In the random oracle model, the protocol Π_{COMMIT} implements $\mathcal{F}_{\text{COMMIT}}$ with computational security against any static, active adversary corrupting at most $n - 1$ parties.*

Proof. We here sketch a simulator such that the environment cannot distinguish if it is playing with the real protocol or the functionality composed with the simulator. Note that the simulator replies to queries to the random oracle $\mathcal{H}$ made by the adversary.

To simulate a Commit call, if the committer P_i is honest, the simulator selects a random value c and gives (c, i, τ_v) to the adversary. Note that (i, τ_v) is given to the simulator by $\mathcal{F}_{\text{COMMIT}}$ hereafter receiving $(\mathsf{Commit}, v, i, \tau_v)$ from P_i. Whereas if the committer is corrupt, then it either queries $\mathcal{H}$ to get c, or it does not query it. Therefore, on receiving (c^*, i, τ_v) from the adversary, the simulator has v (if $\mathcal{H}$ was queried) so it sets $v^* \leftarrow v$. If $\mathcal{H}$ was not queried, the simulator sets dummy input v^* and the internal flag $\mathsf{Abort}_{i,\tau_v}$ to true. It then sends $(\mathsf{Commit}, v^*, i, \tau_v)$ to $\mathcal{F}_{\text{COMMIT}}$.

An Open call is simulated as follows. If the committer is honest, the simulator gets (v, i, τ_v) when P_i inputs $(\mathsf{Open}, i, \tau_v)$ to $\mathcal{F}_{\text{COMMIT}}$. The simulator selects random r and sets $o \leftarrow v||r$. It can now hand (o, i, τ_v) to the adversary. If the random oracle is queried on o, the simulator sends c as response. If the committer is corrupt, the simulator gets (i, τ_v) from the adversary, it checks whether $\mathsf{Abort}_{i,\tau_v}$ is true, if so it sends $(\mathsf{NoOpen}, i, \tau_v)$ to $\mathcal{F}_{\text{COMMIT}}$. Otherwise, the simulator sends $(\mathsf{Open}, i, \tau_i)$ to $\mathcal{F}_{\text{COMMIT}}$.

The adversary will notice that queries to $\mathcal{H}$ are simulated only if o has been queried before resulting in different c, but as r is random this happens only with negligible probability (assuming that the size of the output domain of $\mathcal{H}$ is large enough). Also, in a simulated run, if the adversary does not query $\mathcal{H}$ when committing will result in abort. The

probability that in a real run players do not abort, is equivalent to the the probability that adversary correctly guesses the output of $\mathcal{H}$, which happens with negligible probability.

□

B Key Generation : Protocol, Functionalities and Security Proof

B.1 Protocol

The protocol Π_{KEYGEN}

Initialize:

1. Every player P_i samples a uniform $e_i \leftarrow \{1, \ldots, c\}$ and asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast the handle $\tau_i^e \leftarrow \mathsf{Commit}(e_i)$ for a commitment to e_i.
2. Every player P_i samples a seed $s_{i,j}$ and asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_{i,j}^s \leftarrow \mathsf{Commit}(s_{i,j})$.
3. Every player P_i computes and broadcasts $a_{i,j} \leftarrow \mathcal{U}_{s_{i,j}}(q_1, \phi(m))$.

Stage 1:

4. All the players compute $a_j \leftarrow a_{1,j} + \cdots + a_{n,j}$.
5. Every player P_i computes $\mathfrak{s}_{i,j} \leftarrow \mathcal{HWT}_{s_{i,j}}(h, \phi(m))$ and $\epsilon_{i,j} \leftarrow \mathcal{DG}_{s_{i,j}}(\sigma^2, \phi(m))$, and broadcasts $b_{i,j} \leftarrow [a_j \cdot \mathfrak{s}_{i,j} + p \cdot \epsilon_{i,j}]_{q_1}$.

Stage 2:

6. All the players compute $b_j \leftarrow b_{1,j} + \cdots + b_{n,j}$ and set $\mathfrak{pk}_j \leftarrow (a_j, b_j)$..
7. Every player P_i computes and broadcasts $\mathfrak{enc}'_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}_j}(-p_1 \cdot \mathfrak{s}_{i,j}, \mathcal{RC}_{s_{i,j}}(0.5, \sigma^2, \phi(m)))$.

Stage 3:

8. All the players compute $\mathfrak{enc}'_j \leftarrow \mathfrak{enc}'_{1,j} + \cdots + \mathfrak{enc}'_{n,j}$.
9. Every player P_i computes $\mathfrak{zero}_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}_j}(0, \mathcal{RC}_{s_{i,j}}(0.5, \sigma^2, \phi(m)))$.
10. Every player P_i computes and broadcasts $\mathfrak{enc}_{i,j} \leftarrow (\mathfrak{s}_{i,j} \cdot \mathfrak{enc}'_j) + \mathfrak{zero}_{i,j}$.

Output:

11. All the players compute $\mathfrak{enc}_j \leftarrow \mathfrak{enc}_{1,j} + \cdots + \mathfrak{enc}_{n,j}$ and set $\mathfrak{epk}_j \leftarrow (\mathfrak{pk}_j, \mathfrak{enc}_j)$.
12. Every player P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^e)$. If any opening failed, the players output the numbers of the respective players, and the protocol aborts.
13. All players compute the challenge $\mathsf{chall} \leftarrow 1 + \left(\left(\sum_{i=1}^n e_i\right) \bmod c\right)$.
14. Every player P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_{i,j}^s)$ for $j \neq \mathsf{chall}$. If any opening failed, the players output the numbers of the respective players, and the protocol aborts.
15. All players obtain the values committed, compute all the derived values and check that they are correct.
16. If any of the checks fail, the players output the numbers of the respective players, and the protocol aborts. Otherwise, every player P_i sets
    - $\mathfrak{s}_i \leftarrow \mathfrak{s}_{i,\mathsf{chall}}$,
    - $\mathfrak{pk} \leftarrow (a_{\mathsf{chall}}, b_{\mathsf{chall}})$, $\mathfrak{epk} \leftarrow (\mathfrak{pk}, \mathfrak{enc}_{\mathsf{chall}})$.

Fig. 3. The protocol for key generation.

B.2 Functionality

The ideal functionality $\mathcal{F}_{\text{KEYGEN}}$

Generation:

1. The functionality waits for seeds $\{s_i\}_{i\in A}$ from the adversary, and samples a seed s_i for every honest player P_i.
2. It computes $a_i \leftarrow \mathcal{U}_{s_i}(q_1, \phi(m))$ for all i.
3. It computes $\mathfrak{s}_i \leftarrow \mathcal{HWT}_{s_i}(h, \phi(m))$, $\epsilon_i \leftarrow \mathcal{DG}_{s_i}(\sigma^2, \phi(m))$ and $b_i \leftarrow [a \cdot \mathfrak{s}_i + p \cdot \epsilon_i]_{q_1}$ for all i.
4. It computes $b \leftarrow b_1 + \cdots + b_n$.
5. It computes $\mathfrak{enc}'_i \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(-p_1 \cdot \mathfrak{s}_i, \mathcal{RC}_{s_i}(0.5, \sigma^2, \phi(m)))$ for every P_i.
6. It computes $\mathfrak{enc}' \leftarrow \mathfrak{enc}'_1 + \cdots + \mathfrak{enc}'_n$.
7. It computes $\mathfrak{zero}_i \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(0, \mathcal{RC}_{s_i}(0.5, \sigma^2, \phi(m)))$ for every P_i, and $\mathfrak{enc}_i \leftarrow (\mathfrak{s}_i \cdot \mathfrak{enc}') + \mathfrak{zero}_i$ for all players P_i.
8. It leaks $a_i, b_i, \mathfrak{enc}'_i, \mathfrak{enc}_i$ for all i to the adversary and waits for either Proceed, Cheat, or Abort.

Proceed: The functionality sends $\mathfrak{s}_i$, $\mathfrak{pk} = (a, b)$, $\mathfrak{epk} = (\mathfrak{pk}, \mathfrak{enc})$ to P_i, for all i.

Cheat: On input Cheat, with probability $1 - 1/c$ the functionality leaks NoSuccess and goes to "Abort"; otherwise:

1. It leaks the seeds of the honest parties and sends Success to the adversary.
2. It repeats the following loop:
   - It waits for the adversary to input values a_i^*, (resp. $(\mathfrak{s}_i^*, b_i^*)$, $\mathfrak{enc}_i'^*$, $\mathfrak{enc}_i^*$) for $i \in A$.
   - It overwrites a_i, (resp. $(\mathfrak{s}_i, b_i)$, $\mathfrak{enc}'_i$, $\mathfrak{enc}_i$) for $i \in A$.
   - It recomputes a (resp. b, $\mathfrak{enc}'$, $\mathfrak{enc}$) accordingly.
3. It waits for Proceed or Abort.

Abort:

1. The functionality leaks the seeds of the honest parties if it never did so.
2. It then waits for a set $S \subseteq A$, sends it to the honest players, and aborts.

Fig. 4. The ideal functionality for key generation.

B.3 Proof of Theorem 1

Proof. We build a simulator $\mathcal{S}_{\text{KEYGEN}}$ to work on top of the ideal functionality $\mathcal{F}_{\text{KEYGEN}}$, such that the environment cannot distinguish whether it is playing with the protocol Π_{KEYGEN} and $\mathcal{F}_{\text{COMMIT}}$, or the simulator and $\mathcal{F}_{\text{KEYGEN}}$. The simulator is given in Figure 6.

We now proceed with the analysis of the simulation. Let A denote the set of players controlled by the adversary. In steps 1 and 2 the simulator sends random handles to the adversary, as it would happen in the real protocol. In steps 3–11 the simulation is perfect for all the threads where the simulator knows the seeds of the honest players, since those are generated as in the protocol. In case of no cheat nor abort the simulation is also perfect for the thread where the simulator does not know the seeds of the honest players, since the simulator forwards honest values provided by the functionality. In case of cheating at the thread pointed by chall, the simulator gets the seeds also for the remaining thread and will replace the honestly precomputed intermediate values $a_{i,\mathsf{chall}}$, $\mathfrak{s}_{i,\mathsf{chall}}$, $b_{i,\mathsf{chall}}$, $\mathfrak{enc}'_{i,\mathsf{chall}}$, $\mathfrak{enc}_{i,\mathsf{chall}}$ with the ones compatible with the deviation of the adversary – the honest values computed after a cheat reflect the adversarial behaviour of the real protocol, so a simulated run is again indistinguishable from a real run of the protocol.

Steps 12, 14 are statistically indistinguishable from a protocol run, since the simulator plays also the role of $\mathcal{F}_{\text{COMMIT}}$.

Step 15 needs more work: we need to ensure that the success probability in a simulated run is the same as the one in a real run of the protocol. If the adversary does not deviate, the protocol succeeds. The same applies for a simulated run, since the simulator goes through "Pass" at every stage. More in detail, the simulator sampled and computed all the values at the threads not pointed by the challenge as in a honest run of the protocol, while values at the thread pointed by the challenge are correctly evaluated and sent to the honest players by $\mathcal{F}_{\text{KEYGEN}}$. In case the adversary cheats only on one thread, in a real execution of the protocol the adversary succeeds in the protocol with probability $1/c$; the same holds in a simulated run, since the simulator goes through Cheat in CheatSwitch once and the functionality leaks

Success, which happens with the same probability, and later the simulator will not abort. If the adversary deviates on more than one branch (considering all stages), both the real protocol and the simulation will abort at step 15.

Finally, if the protocol aborts due to failure at opening commitments, both the functionality and the players output the numbers of corrupted players who failed to open their commitments. If the protocol aborts at step 15, the output is the numbers of players who deviated in threads other than chall in both the functionality and the protocol. □

The procedure CheatSwitch

Pass: Checks passed for all threads or there was a successful cheat in earlier stages and checks now passed for at least all threads except for the one pointed by chall.
- The simulator continues.

Cheat: There was no cheat in previous stages and all checks now passed except for the ones pointed by a unique thread j.
- The simulator adds (j, P_i) to a list L, for all players P_i making the check not pass.
- The simulator sends Cheat to $\mathcal{F}_{\text{KEYGEN}}$, and gets and stores every honest seeds $s_{i,\mathsf{chall}}$.
- If the functionality sends NoSuccess:
  - If $j = \mathsf{chall}$, it resamples a different $\mathsf{chall} \leftarrow \{1, \ldots, c\} \setminus \{\mathsf{chall}\}$.

  It continues the simulation according to the protocol.
- If the functionality sends Success: the simulator sets $\mathsf{chall} = j$ and continues according to the protocol.

Abort: In more than one thread checks did not pass (counting also checks in previous stages).
- The simulator adds (j, P_i) to a list L, for all branches j and players P_i making the check not pass.
- The simulator sends Abort to $\mathcal{F}_{\text{KEYGEN}}$ and it continues the rest of the simulation according to the protocol.

Fig. 5. The cheat switch.

The simulator $\mathcal{S}_{\text{KEYGEN}}$

Initialize:

- In Step 1 the simulator obtains e_i by every corrupt P_i, and broadcasts τ_i^e as $\mathcal{F}_{\text{COMMIT}}$ would do. It samples chall uniformly in $\{1, \ldots, c\}$, and it broadcasts a handle τ_i^e for every honest P_i.
- In Step 2 the simulator sees the random values $s_{i,j}$ for $i \in A$.
  It inputs $\{s_{i,\mathsf{chall}}\}_{i \in A}$ to $\mathcal{F}_{\text{KEYGEN}}$, therefore obtaining a full transcript of the thread corresponding to chall.
  For the threads $j \neq \mathsf{chall}$, for honest P_i, the simulator samples $s_{i,j}$ honestly and broadcasts a handle $\tau_{i,j}^s$ for every honest P_i for every thread.
- In Step 3, the simulator computes $a_{i,j}$ honestly for $i \notin A$ and $j \neq \mathsf{chall}$, while it defines $a_{i,\mathsf{chall}}$ for $i \notin A$ as the values a_i obtained from the transcript given by $\mathcal{F}_{\text{KEYGEN}}$.
  It then broadcasts $a_{i,j}$ for honest P_i and waits for broadcasts $a_{i,j}$ by the corrupt players, and it checks $a_{i,j} = \mathcal{U}_{s_{i,j}}(q, \phi(m))$ for all dishonest P_i. For this check the simulator enters $\mathsf{CheatSwitch}$. If there was a successful cheat on the thread pointed by chall, the simulator inputs $a_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KEYGEN}}$ for $i \in A$.

Stage 1:

- In Step 4 the simulator acts as in the protocol.
- In Step 5 for all the honest seeds that are known by the simulator, the simulator computes $b_{i,j}$ honestly for $i \notin A$.
  If the simulator does not know the seeds $s_{i,\mathsf{chall}}$ for honest P_i, it defines $b_{i,\mathsf{chall}}$ for $i \notin A$ as the values b_i obtained from the transcript given by $\mathcal{F}_{\text{KEYGEN}}$.
  It then broadcasts $b_{i,j}$ for honest P_i and waits for broadcasts $b_{i,j}$ by the corrupt players. It then checks $b_{i,j} = [a_{\mathsf{chall}} \cdot \mathcal{HWT}_{s_{i,\mathsf{chall}}}(h, \phi(m)) + p \cdot \mathcal{RC}_{s_{i,\mathsf{chall}}}(\sigma^2, \phi(m))]_{q_1}$ for all dishonest P_i. For this check the simulator enters $\mathsf{CheatSwitch}$. If there was a successful cheat on the thread pointed by chall, the simulator inputs $(\mathfrak{s}_{i,\mathsf{chall}}, b_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KEYGEN}}$ for $i \in A$.

Stage 2:

- In Step 6 the simulator acts as in the protocol.
- In Step 7 for all the honest seeds that are known by the simulator, the simulator computes $\mathfrak{enc}'_{i,j}$ honestly for $i \notin A$.
  If the simulator does not know the seeds $s_{i,\mathsf{chall}}$ for honest P_i, it defines $\mathfrak{enc}'_{i,\mathsf{chall}}$ for $i \notin A$ as the values $\mathfrak{enc}'_i$ obtained from the transcript given by $\mathcal{F}_{\text{KEYGEN}}$.
  It then broadcasts $\mathfrak{enc}'_{i,j}$ for honest P_i and waits for broadcasts $\mathfrak{enc}'_{i,j}$ by the corrupt players. It then checks $\mathfrak{enc}'_{i,j} = \mathsf{Enc}_{\mathfrak{pk}}(-p_1 \cdot \mathfrak{s}_{i,j}, \mathcal{RC}_{s_{i,j}}(0.5, \sigma^2, \phi(m)))$ for all dishonest P_i. For this check the simulator enters $\mathsf{CheatSwitch}$. If there was a successful cheat on the thread pointed by chall, the simulator inputs $\mathfrak{enc}'_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KEYGEN}}$ for $i \in A$.

Stage 3:

- In Step 8, 9 the simulator acts as in the protocol.
- In Step 10 for all the honest seeds that are known by the simulator, the simulator computes $\mathfrak{enc}_{i,j}$ honestly for $i \notin A$.
  If the simulator does not know the seeds $s_{i,\mathsf{chall}}$ for honest P_i, it defines $\mathfrak{enc}_{i,\mathsf{chall}}$ for $i \notin A$ as the values $\mathfrak{enc}_i$ obtained from the transcript given by $\mathcal{F}_{\text{KEYGEN}}$.
  It then broadcasts $\mathfrak{enc}_{i,j}$ for honest P_i and waits for broadcasts $\mathfrak{enc}_{i,j}$ by the corrupt players. It then checks $\mathfrak{enc}_{i,j} = (\mathfrak{s}_{i,j} \cdot \mathfrak{enc}'_{i,j}) + \mathfrak{zero}_{i,j}$ for all dishonest P_i. For this check the simulator enters $\mathsf{CheatSwitch}$. If there was a successful cheat on the thread pointed by chall, the simulator inputs $\mathfrak{enc}_{i,\mathsf{chall}}$ to $\mathcal{F}_{\text{KEYGEN}}$ for $i \in A$.

Output:

- Step 11 is performed according to the protocol.
- The simulator samples e_i for $i \notin A$ uniformly such that $1 + \left(\left(\sum_{i=1}^n e_i\right) \bmod c\right) = \mathsf{chall}$.
- Step 12 is performed according to the protocol, but the simulator opens τ_i^e revealing the values e_i for all honest P_i, and if the check fails the simulator sends Abort to $\mathcal{F}_{\text{KEYGEN}}$ and inputs the set of all players failing in opening.
- Step 13 is performed according to the protocol.
- Step 14 is performed according to the protocol, and if the check fails the simulator sends Abort to $\mathcal{F}_{\text{KEYGEN}}$ and inputs the set of all players failing in opening.
- Step 15 is performed according to the protocol, and the simulator defines

$$S = \{i \in A \mid (j, P_i) \in L;\ j \in \{1, \ldots, c\};\ j \neq \mathsf{chall}\},$$

  i.e. the set of corrupt players who cheated at any thread different from chall.
  - If $S \neq \varnothing$ (i.e. cheats at a thread which is going to be opened), the simulator sends Abort to $\mathcal{F}_{\text{KEYGEN}}$ and inputs S.
  - If $S = \varnothing$ (i.e. successful or no cheats), the simulator sends $\mathsf{Proceed}$ to $\mathcal{F}_{\text{KEYGEN}}$.
- Step 16 is performed according to the protocol.

Fig. 6. The simulator for the key generation functionality.

B.4 Semantic security of $\mathcal{F}_{\text{KEYGEN}}$

Here we prove the semantic security of the cryptosystem resulting from an execution of $\mathcal{F}_{\text{KEYGEN}}$, based on the ring-LWE problem and a form of KDM security for quadratic functions. The ring-LWE assumption we use takes an extra parameter h, as our scheme chooses binary, low hamming weight, secret keys for better efficiency and parameter sizes, but note that the results here also apply to secrets drawn from other distributions.

Definition 1 (Decisional Ring Learning With Errors assumption). *The single sample decisional ring-LWE assumption* $\mathsf{RLWE}_{q,\sigma^2,h}$ *states that*

$$(a, a \cdot s + e) \stackrel{c}{\approx} (a, u)$$

where $s \leftarrow \mathcal{HWT}(h, \phi(m))$, $e \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$ *and* a, u *are uniform over* R_q.

The KDM security assumption, below, can be viewed as a distributed extension to the usual key switching assumption for FHE schemes. In this case we need 'encryptions' of quadratic functions of *additive shares* of the secret key to remain secure. Note that whilst it is easy to show KDM security for *linear* functions of the secret [7], it is not known how to extend this to the functions required here without increasing the length of ciphertexts.

Definition 2 (KDM security assumption). *If* $\mathfrak{s}_i \leftarrow \mathcal{HWT}(h)$, $\mathfrak{s} = \sum_{i=0}^{n-1} \mathfrak{s}_i$ *and* f *is any degree 2 polynomial then*

$$(a, a \cdot \mathfrak{s} + p \cdot e + f(\mathfrak{s}_0, \ldots, \mathfrak{s}_{n-1})) \stackrel{c}{\approx} (a, a \cdot \mathfrak{s} + p \cdot e)$$

where $a, u \leftarrow \mathcal{U}(q, \phi(m))$, $e \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$.

The following lemma states that distinguishing any number of 'amortized' ring-LWE samples with different, independent, secret keys but common first component a, from uniform is as hard as distinguishing just one ring-LWE sample from uniform. It was proven for the (standard) LWE setting with $n = 3$ in [23]; here we need a version with ring-LWE for any n.

Lemma 3 (Adapted from [23, Lemma 7.6]). *Suppose* $a, u_i \leftarrow \mathcal{U}(q, \phi(m))$, $\mathfrak{s}_i \leftarrow \mathcal{HWT}(h, \phi(m))$ *and* $e_i \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$ *for* $i = 0, \ldots, n-1$, $n \in \mathbb{N}$. *Then*

$$\{(a, a \cdot \mathfrak{s}_i + e_i)\}_i \stackrel{c}{\approx} \{(a, u_i)\}_i$$

under the single sample ring-LWE assumption $\mathsf{RLWE}_{q,\sigma^2,h}$.

Proof. Suppose an adversary $\mathcal{A}$ can distinguish between the above distributions with non-negligible probability. We construct an adversary $\mathcal{B}$ that solves the RLWE problem. Given a challenge (a, b) from the RLWE oracle, $\mathcal{B}$ sets $b_0 = b$ and $b_i = a \cdot \mathfrak{s}_i + e_i$ for $i = 1, \ldots, n-1$, where $e_i \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$, $\mathfrak{s}_i \leftarrow \mathcal{HWT}(h, \phi(m))$. $\mathcal{B}$ sends all pairs (a, b_i) to $\mathcal{A}$ and returns the output of $\mathcal{A}$ in response to the challenge.

Since the values (a, b_i) for $i = 1, \ldots, n-1$ are all valid amortized ring-LWE samples, the only difference between the view of $\mathcal{A}$ and that of a real set of inputs is b_0, and so the advantage of $\mathcal{B}$ in solving $\mathsf{RLWE}_{q,\sigma^2,h}$ is exactly that of $\mathcal{A}$ in solving the amortized ring-LWE problem with n samples. □

Theorem 6 (restatement of Theorem 2). *If the functionality* $\mathcal{F}_{\text{KEYGEN}}$ *is used to produce a public key* $\mathfrak{epk}$ *and secret keys* $\mathfrak{s}_i$ *for* $i = 0, \ldots, n-1$ *then the resulting cryptosystem is semantically secure based on the hardness of* $\mathsf{RLWE}_{q_1,\sigma^2,h}$ *and the KDM security assumption.*

Proof. Suppose there is an adversary $\mathcal{A}$ that can interact with $\mathcal{F}_{\text{KEYGEN}}$ and distinguish the public key $(\mathfrak{pk}, \mathfrak{epk})$ from uniform. We construct an algorithm $\mathcal{B}$ that distinguishes amortized ring-LWE samples from uniform. By Lemma 3 this is at least as hard as breaking single sample ring-LWE. If the public key is pseudorandom then semantic security of encryption easily follows, as ciphertexts are just ring-LWE samples. Note that we only consider a non-cheating adversary – if $\mathcal{A}$ cheats then it can trivially break the scheme with non-negligible probability $1/c$.

The challenger gives $\mathcal{B}$ the values $a_c, b_{c,0}, \ldots, b_{c,n-1}$. $\mathcal{B}$ must now simulate an execution of $\mathcal{F}_{\text{KEYGEN}}$ with $\mathcal{A}$ to determine whether the challenge is uniform or of the form $(a_c, a_c \cdot \mathfrak{s}_i + e_i)$ for $\mathfrak{s}_i \leftarrow \mathcal{HWT}(h, \phi(m))$ and $e_i \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$.

To start with we receive the adversary's seeds s_i for every corrupt player $i \in A$. We must then simulate the values $a_i, b_i, \mathfrak{enc}'_i, \mathfrak{enc}_i$ (for all i) that are leaked to the adversary in $\mathcal{F}_{\text{KEYGEN}}$. For corrupt players we simply compute these values according to $\mathcal{F}_{\text{KEYGEN}}$ using the adversary's seeds. Next we have to simulate the honest players' values, which we do using the challenge $a_c, b_{c,0}, \ldots, b_{c,n-1}$. First we scale the challenge by p, so that it takes the form $(a_c, a_c \cdot \mathfrak{s}_i + p \cdot e_i)$ if they are genuine RLWE samples. Since p is coprime to q this still has the same distribution as the original challenge.

Now $\mathcal{B}$ calculates uniform consistent shares $a_{c,i}$, for every honest player P_i, of a_c, and sends $\mathcal{A}$ the pairs $a_{c,i}, b_{c,i}$. If the challenge values are amortized ring-LWE samples, then these are consistent with the pairs (a_i, b_i) computed by $\mathcal{F}_{\text{KEYGEN}}$, since a_i is uniform and $b_i = a_i \cdot \mathfrak{s}_i + e_i$.

Next, $\mathcal{B}$ must provide $\mathcal{A}$ with simulations of players' contributions to the key-switching data $\mathfrak{enc}'_i, \mathfrak{enc}_i$ for all honest players P_i. For both of these sets of values, $\mathcal{B}$ simply re-randomizes the pair (a_c, b_c) and sends this to $\mathcal{A}$. This can be done by, for example, computing an encryption of zero under the public key (a_c, b_c) (where $b_c = \sum_i b_{c,i}$). Notice that $\mathfrak{enc}'_i$ is just an encryption of $-p_1 \cdot \mathfrak{s}_i$ under the public key (a, b), and so by the KDM security assumption is (perfectly) indistinguishable from a re-randomized version of (a, b). For $\mathfrak{enc}_i$, recall that $\mathcal{F}_{\text{KEYGEN}}$ computes $\mathfrak{enc}_i = \mathfrak{s}_i \cdot \mathfrak{enc}' + \mathfrak{zero}_i$. Now writing $\mathfrak{zero}_i = (a \cdot v_i + p \cdot e_{0,i}, b \cdot v_i + p \cdot e_{1,i})$ and $\mathfrak{enc}' = (a \cdot v + p \cdot e_0, b \cdot v + p \cdot e_1 - p_1 \cdot \mathfrak{s})$, we see that

$$
\begin{aligned}
\mathfrak{enc}_i &= (a \cdot v \cdot \mathfrak{s}_i + a \cdot v_i + p \cdot (e_0 \cdot \mathfrak{s}_i + e_{0,i}), b \cdot v \cdot \mathfrak{s}_i + b \cdot v_i + p \cdot (e_1 \cdot \mathfrak{s}_i + e_{1,i}) - p_1 \cdot \mathfrak{s} \cdot \mathfrak{s}_i) \\
&= \left(\underbrace{a \cdot (v \cdot \mathfrak{s}_i + v_i)}_{a'_i} + p \cdot \underbrace{(e_0 \cdot \mathfrak{s}_i + e_{0,i})}_{e'_{0,i}}, \underbrace{a \cdot (v \cdot \mathfrak{s}_i + v_i)}_{a'_i} \cdot \mathfrak{s} + p \cdot \underbrace{(e_1 \cdot \mathfrak{s}_i + e_{1,i} + e)}_{e'_{1,i}} - p_1 \cdot \mathfrak{s} \cdot \mathfrak{s}_i \right) \\
&= \left(a'_i + p \cdot e'_{0,i}, a'_i \cdot \mathfrak{s} + p \cdot e'_{1,i} - p_1 \cdot \mathfrak{s} \cdot \mathfrak{s}_i \right).
\end{aligned}
$$

Notice that the first component of $\mathfrak{enc}_i$ corresponds to the second half of a ring-LWE sample $(a, a \cdot (v \cdot \mathfrak{s}_i + v_i) + p \cdot e_{0,i})$ with secret $v \cdot \mathfrak{s}_i + v_i$. The second component of $\mathfrak{enc}_i$ corresponds to a ring-LWE sample with secret $\mathfrak{s}$ and first half a'_i, with an added quadratic function of the key $-p_1 \cdot \mathfrak{s} \cdot \mathfrak{s}_i$. By the KDM security assumption, this is indistinguishable from a genuine ring-LWE sample, so $\mathfrak{enc}_i$ can also be perfectly simulated by re-randomizing (a_c, b_c).

To finish the simulated execution of $\mathcal{F}_{\text{KEYGEN}}$, $\mathcal{B}$ sends $\mathcal{A}$ shares of the secret key for all P_i where $i \in A$ (i.e. all dishonest players), by sampling randomness using the seeds that were provided to $\mathcal{B}$ at the beginning. $\mathcal{B}$ then waits for $\mathcal{A}$ to give an answer and returns this in response to the challenger. Notice that throughout the simulation, all values passed to $\mathcal{A}$ were ring-LWE samples derived from the challenge $(a_c, b_{0,c}, \ldots, b_{n-1,c})$. We showed that if the challenge is an amortized ring-LWE sample then $\mathcal{A}$'s input is indistinguishable from the output of $\mathcal{F}_{\text{KEYGEN}}$, whereas if the challenge is uniform then so is $\mathcal{A}$'s input. Therefore if $\mathcal{A}$ is successful in distinguishing the resulting public key from uniform then $\mathcal{A}$ must have solved the ring-LWE challenge.

□

C EncCommit: Protocol, Functionalities and Security Proofs

C.1 Protocol

Protocol $\Pi_{\text{EncCommit}}$

Usage: The specific distribution of the message is defined by the input parameter cond. The output is a single message $\mathbf{m}_i$ private to each player, and a public ciphertext $\mathfrak{c}_i$ from player i. The protocol runs in two phases; a commitment phase and an opening phase.

KeyGen: The players execute Π_{KeyGen} to obtain $\mathfrak{s}_i$, $\mathfrak{pk}$, and $\mathfrak{epk}$.

Commitment Phase:

1. Every player P_i samples a uniform $e_i \leftarrow \{1, \ldots, c\}$, and queries $\mathsf{Commit}(e_i)$ to $\mathcal{F}_{\text{Commit}}$, which broadcasts a handle τ_i^e.
2. For $j = 1, \ldots, c$
   (a) Every player P_i samples a seed $s_{i,j}$ and queries $\mathsf{Commit}(s_{i,j})$ to $\mathcal{F}_{\text{Commit}}$, which broadcasts a handle $\tau_{i,j}^s$.
   (b) Every player P_i generates $\mathbf{m}_{i,j}$ according to cond using $\mathsf{PRF}_{s_{i,j}}$.
   (c) Every player P_i computes and broadcasts $\mathfrak{c}_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m}_{i,j})$ using $\mathsf{PRF}_{s_{i,j}}$ to generate the randomness.
3. Every player P_i calls $\mathcal{F}_{\text{Commit}}$ with $\mathsf{Open}(\tau_i^e)$. All players get e_i. If any opening failed, the players output the numbers of the respective players, and the protocol aborts.
4. All players compute $\mathsf{chall} \leftarrow 1 + \left(\left(\sum_{i=1}^n e_i\right) \bmod c\right)$.

Opening Phase:

5. Every player P_i calls $\mathcal{F}_{\text{Commit}}$ with $\mathsf{Open}(\tau_{i,j}^s)$ for all $j \neq \mathsf{chall}$ so that all players obtain the value $s_{i,j}$ for $j \neq \mathsf{chall}$. If any opening fails, the players output the numbers of the respective players, and the protocol aborts.
6. For all $j \neq \mathsf{chall}$ and all $i' \leq n$, the players check whether $\mathfrak{c}_{i',j}$ was generated correctly using $s_{i',j}$. If not, they output the numbers of the respective players i', and the protocol aborts.
7. Otherwise, every player P_i stores $\{\mathfrak{c}_{i',\mathsf{chall}}\}_{i' \leq n}$ and $m_{i,\mathsf{chall}}$.

Fig. 7. Protocol that allows ciphertext to be used as commitments for plaintexts

C.2 Functionalities

C.3 Proof of Theorem 3

Proof. We construct a simulator $\mathcal{S}_{\text{SHE}}$ (see Figure 9) working on top of $\mathcal{F}_{\text{SHE}}$ such that the environment can not distinguish whether it is playing with the real protocol $\Pi_{\text{EncCommit}}$ and $\mathcal{F}_{\text{KeyGen}}$ or with $\mathcal{F}_{\text{SHE}}$ and $\mathcal{S}_{\text{SHE}}$. The simulator is given in Figure 9.

Calls to $\mathcal{F}_{\text{KeyGen}}$ are simulated as in $\mathcal{S}_{\text{KeyGen}}$. We now focus on the commitment phase.

Let A be the set of indices of corrupted players. The simulator starts assuming that the adversary will behave honestly. It samples a uniform $j_0 \leftarrow \{1, \ldots, c\}$ and seeds $\{s_{i,j}\}_{i \notin A, j \neq j_0}$. If the adversary does not deviate, then round j_0 will remain unopened, otherwise the simulator will have to adjust this. We can simulate each round j as follows. First, the simulator gets corrupted seeds $s_{i,j}$ for $i \in A$ when the adversary commits to them in step 2a. It gives in return random handles $\tau_{i,j}^s$ on behalf of each honest player P_i.

If $j \neq j_0$, the simulator engages with the adversary in a normal run of steps 2b to 2c using seeds $s_{i,j}$ for honest player P_i. Since the simulator knows the corrupt seeds of the current round j, it can check whether the adversary behaved honestly. If the adversary did not, then the simulator stores index j in the cheating list.

If $j = j_0$, the simulator checks again whether the adversary computed the right encryptions $\{\mathfrak{c}_i\}_{i \in A}$. If it did not, the simulator stores j_0 in the cheating list. Then the simulator calls EncCommit to $\mathcal{F}_{\text{SHE}}$ on seeds $\{s_{i,j_0}\}_{i \in A}$ and gets back $\{\mathfrak{c}_i\}_{i \notin A}$, which are the values computed by the functionality. It then sets $\mathfrak{c}_{i,j_0} \leftarrow \mathfrak{c}_i$ and pass them onto the adversary in step 2c.

Once the last round is finished, the simulator checks the cheating list. There are three possibilities:

The ideal functionality $\mathcal{F}_{\text{SHE}}$

Usage: The functionality is split into a one-run stage which computes the key material and a stage which can be accessed several times and is targeted to replace the zero-knowledge protocols in [14].

KeyGen: On input KeyGen the functionality acts as a copy of $\mathcal{F}_{\text{KEYGEN}}$.
Notice that all the variables used during this call are available for later use.

EncCommit: On input EncCommit the functionality does the following.

Initialize: Denote by A the set of indices of corrupt players. On input Start by all players, sample, at random, seeds $\{s_i\}_{i\notin A}$ and wait for corrupted seeds $\{s_i\}_{i\in A}$ from the adversary.

Computation:

1. It sets $\mathbf{m}_i \leftarrow \mathsf{PRF}_{s_i}$ subject to condition cond.
2. It sets $\mathfrak{c}_i = \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m}_i, \mathcal{RC}_{s_i}(0.5, \sigma^2, \phi(m)))$ for each player P_i.
3. It gives $\{\mathfrak{c}_i\}_{i\notin A}$ to the adversary, and waits for signal Deliver, Cheat or Abort.

Delivery: The functionality sends $\mathbf{m}_i, \{\mathfrak{c}_j\}_{j\leq n}$ to player P_i.

Cheat: The functionality gives $\{s_i\}_{i\notin A}$ to the adversary, then it decides to do either of the following things:

- With probability $1/c$ it sends Success to the adversary, it waits for $\{\mathbf{m}_i, \mathfrak{c}_i\}_{i\in A}$, and outputs $\mathbf{m}_i, \{\mathfrak{c}_j\}_{i\leq n}$ to player P_i.
- Otherwise sends NoSuccess to the adversary, and goes to abort.

Abort: The functionality waits for the adversary to input $S \subseteq A$, and outputs S to all players.

Fig. 8. The ideal functionality for key generation and $\Pi_{\text{ENCCOMMIT}}$.

- The list is empty. In other words, the adversary behaved honestly. The simulator sets $\mathsf{chall} \leftarrow j_0$, and sends Deliver to the functionality if all commitments are successfully opened. The output of $\mathcal{F}_{\text{SHE}}$ and what the adversary has already seen seen will be consistent since $\mathcal{F}_{\text{SHE}}$ was called in round j_0 with the right seeds $\{s_{i,j_0}\}_{i\in A}$.
- The list contains only one index j_1. In this case the simulator sends Cheat and gets in return seeds $\{s_i\}_{i\notin A}$ used by the functionality. It sets $s_{i,j_0} \leftarrow s_i$ for each honest player P_i. It then waits for the answer.
  - If the functionality returns Success, the simulator has to make the adversary believe that round j_1 will remain unopened. It sets $\mathsf{chall} \leftarrow j_1$. If all commitments are successfully opened, it sends $\{\mathbf{m}_{i,j_1}, \mathfrak{c}_{i,j_1}\}_{i\in A}$ to the functionality in order to make consistent players' outputs and what the adversary has already seen.
  - If it returns NoSuccess, the simulator has to make the adversary believe that round j_1 will be opened. Therefore it samples $\mathsf{chall} \leftarrow \{1, \ldots, c\} \setminus \{j_1\}$.
- The list contains at least two indices j_1, j_2. In this case the real protocol would result in abort, so the simulator sends Abort to the functionality and sets $\mathsf{chall} \leftarrow j_0$.

Later the simulator generates the value e_i for each honest player such that $1 + ((\sum_{i=1}^{n} e_i) \bmod c) = \mathsf{chall}$. This ensures that once the challenge is computed, it will point to a round in the same fashion as the protocol would do. Moreover, opening τ_i^e to (any) e_i is does not give clues to the adversary if it is playing in a real run of the protocol or in a simulated one.

In the opening phase, the simulator gives $\{e_i\}_{i\notin A}$ and honest share $\{s_{i,j}\}_{i\notin A, j\neq \mathsf{chall}}$ to the adversary, and if it there was a cheating with no success, then it also sends Abort on behalf of each honest player.

It is clear, from the construction of $\mathcal{F}_{\text{SHE}}$, that all the messages generated by the simulator are indistinguishable from a real run of the protocol. The simulator does the same computations, except in round j_0 where the computation is done by the functionality, and the values are then passed onto the simulator, which forwards them to the adversary.

Finally, if the protocol aborts due to failure at opening commitments, both the functionality and the players output the numbers of corrupted players who failed to open their commitments. If the protocol aborts at step 6, the output is the numbers of players who deviated in threads other than chall in both the functionality and the protocol. □

The simulator $\mathcal{S}_{\text{SHE}}$

KeyGen: $\mathcal{S}_{\text{SHE}}$ acts as $\mathcal{S}_{\text{KEYGEN}}$, but $\mathcal{S}_{\text{SHE}}$ calls $\mathcal{F}_{\text{SHE}}$ on query KeyGen, when $\mathcal{S}_{\text{KEYGEN}}$ would have called $\mathcal{F}_{\text{KEYGEN}}$.

Commitment Phase:

- The simulator chooses random $j_0 \leftarrow \{1, \ldots, c\}$ and seeds $\{s_{i,j}\}_{i \notin A, j \neq j_0}$.
- Acting as the $\mathcal{F}_{\text{COMMIT}}$ functionality, in response to query in step 1 and 2a, for $j = 1, \ldots, c$ the simulator samples $s_{i,j}$ according to the protocol for $i \notin A$ and returns random handles $\{\tau_i^e\}_{i \leq n}, \{\tau_{i,j}^s\}_{i \leq n}$.
- For $j = 1, \ldots, c$, the simulator does the following:
  - If $j \neq j_0$, it performs steps 2b and 2c according to protocol using honest seeds $s_{i,j}$ for each $i \notin A$.
  - If $j = j_0$, it calls $\mathcal{F}_{\text{SHE}}$ on query EncCommit on corrupted seeds $\{s_{i,j_0}\}_{i \in A}$ and gets back honest encryptions $\{\mathfrak{c}_i\}_{i \notin A}$. It then sets $\mathfrak{c}_{i,j_0} \leftarrow \mathfrak{c}_i$ for each $i \notin A$.
- In step 2c, the simulator receives encryptions $\mathfrak{c}_{i,j}^*$ for each $i \in A$ and $j \in \{1, \ldots, c\}$. It generates $\mathbf{m}_{i,j}$ subject to cond, and $\mathfrak{c}_{i,j} \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m}_{i,j})$, and checks if $\mathfrak{c}_{i,j} = \mathfrak{c}_{i,j}^*$. If the equality does not hold, it stores j in a (cheating) list.
- The simulator reads the cheating list. There are three possibilities:
  - The list is empty. The simulator sets chall $\leftarrow j_0$.
  - The list contains only one index j_1. The simulator sends Cheat to $\mathcal{F}_{\text{SHE}}$ and gets $\{s_i\}_{i \notin A}$ back. It then sets $s_{i,j_0} \leftarrow s_i$ for each $i \notin A$.
    - If the functionality returns Success, the simulator sets chall $\leftarrow j_1$.
    - If the functionality returns NoSuccess, the simulator samples chall $\leftarrow \{1, \ldots, c\} \setminus \{j_1\}$.
  - The list contains at least two indices. The simulator sends Abort to $\mathcal{F}_{\text{SHE}}$, gets $\{s_i\}_{i \notin A}$ and sets $s_{i,j_0} \leftarrow s_i$ for each $i \notin A$, and chall $\leftarrow j_0$.
- For all honest P_i the simulator sets e_i uniformly in $1, \ldots, c$ with the constraint $1 + \left(\left(\sum_{i=1}^n e_i\right) \bmod c\right) = \text{chall}$.
- In step 3, the simulator opens the handle τ_i^e to the freshly defined value e_i, for all honest P_i. If the adversary fails to open some of the commitments of corrupted players, the simulator sends Abort and the numbers of the respective players to $\mathcal{F}_{\text{SHE}}$, and it stops.
- Step 4 is performed according to the protocol.

Opening Phase:

- In step 5, the simulator opens the handle $\tau_{i,j}^s$ to $s_{i,j}$ for all honest players $i \notin A$ and $j \neq \text{chall}$. If the adversary fails to open some of the commitments of corrupted players, the simulator sends Abort and the numbers of the respective players to $\mathcal{F}_{\text{SHE}}$, and it stops.
- If the cheating list is empty, the simulator sends Deliver to $\mathcal{F}_{\text{SHE}}$.
- If the functionality returned Success earlier, the simulator inputs $\{\mathbf{m}_{i,\text{chall}}, \mathfrak{c}_{i,\text{chall}}^*\}_{i \in A}$ to the functionality.
- If the functionality returned NoSuccess, or if the cheating list has at least two indices, the simulator inputs to the functionality the number of players $i \in A$ whose $\mathfrak{c}_{i,j}^*$ were computed incorrectly for some $j \neq \text{chall}$.

Fig. 9. The simulator for $\mathcal{F}_{\text{SHE}}$

D Offline Phase : Protocol, Functionalities and Simulators

D.1 Protocols

Protocol MACCheck

Usage: Each player has input α_i and $(\gamma(a_j)_i)$ for $j = 1, \ldots, t$. All players have a public set of opened values $\{a_1, \ldots, a_t\}$; the protocol either succeeds or outputs failure if an inconsistent MAC value is found.

$\mathsf{MACCheck}(\{a_1, \ldots, a_t\})$**:**

1. Every player P_i samples a seed s_i and asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_i^s \leftarrow \mathsf{Commit}(s_i)$.
2. Every player P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^s)$ and all players obtain s_j for all j.
3. Set $s \leftarrow s_1 \oplus \cdots \oplus s_n$.
4. Players sample a random vector $\mathbf{r} = \mathcal{U}_s(p, t)$; note all players obtain the same vector as they have agreed on the seed s.
5. Each player computes the public value $a \leftarrow \sum_{j=1}^{t} r_j \cdot a_j$.
6. Player i computes $\gamma_i \leftarrow \sum_{j=1}^{t} r_j \cdot \gamma(a_j)_i$, and $\sigma_i \leftarrow \gamma_i - \alpha_i \cdot a$.
7. Player i asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_i^\sigma \leftarrow \mathsf{Commit}(\sigma_i)$.
8. Every player calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^\sigma)$, and all players obtain σ_j for all j.
9. If $\sigma_1 + \cdots + \sigma_n \neq 0$, the players output $\varnothing$ and abort.

Fig. 10. Method To Check MACs On Partially Opened Values

Protocol Reshare

Usage: Input is $\mathfrak{c}_{\mathbf{m}}$, where $\mathfrak{c}_{\mathbf{m}} = \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m})$ is a public ciphertext and a parameter enc, where $enc = \mathsf{NewCiphertext}$ or $enc = \mathsf{NoNewCiphertext}$. Output is a share $\mathbf{m}_i$ of $\mathbf{m}$ to each player P_i; and if $enc = \mathsf{NewCiphertext}$, a ciphertext $\mathfrak{c}'_{\mathbf{m}}$. The idea is that $\mathfrak{c}_{\mathbf{m}}$ could be a product of two ciphertexts, which Reshare converts to a "fresh" ciphertext $\mathfrak{c}'_{\mathbf{m}}$. Since Reshare uses distributed decryption (that may return an incorrect result), it is not guaranteed that $\mathfrak{c}_{\mathbf{m}}$ and $\mathfrak{c}'_{\mathbf{m}}$ contain the same value, but it *is* guaranteed that $\sum_i \mathbf{m}_i$ is the value contained in $\mathfrak{c}'_{\mathbf{m}}$.

$\mathsf{Reshare}(\mathfrak{c}_{\mathbf{m}}, enc)$:

1. The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that player i obtains plaintext $\mathbf{f}_i$ and all players obtain $\mathfrak{c}_{\mathbf{f}_i}$, an encryption of $\mathbf{f}_i$.
2. The players compute $\mathfrak{c}_{\mathbf{f}} \leftarrow \mathfrak{c}_{\mathbf{f}_1} + \cdots + \mathfrak{c}_{\mathbf{f}_n}$, and $\mathfrak{c}_{\mathbf{m}+\mathbf{f}} \leftarrow \mathfrak{c}_{\mathbf{m}} + \mathfrak{c}_{\mathbf{f}}$. We define $\mathbf{f} = \mathbf{f}_1 + \cdots + \mathbf{f}_n$, although no party can compute $\mathbf{f}$.
3. The players invoke Protocol DistDec to decrypt $\mathfrak{c}_{\mathbf{m}+\mathbf{f}}$ and thereby obtain $\mathbf{m} + \mathbf{f}$.
4. P_1 sets $\mathbf{m}_1 \leftarrow \mathbf{m} + \mathbf{f} - \mathbf{f}_1$, and each player P_i ($i \neq 1$) sets $\mathbf{m}_i \leftarrow -\mathbf{f}_i$.
5. If $enc = \mathsf{NewCiphertext}$, all players set $\mathfrak{c}'_{\mathbf{m}} \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m} + \mathbf{f}) - \mathfrak{c}_{\mathbf{f}_1} - \cdots - \mathfrak{c}_{\mathbf{f}_n}$, where a default value for the randomness is used when computing $\mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m} + \mathbf{f})$.

Fig. 11. The Protocol For Additively Secret Sharing A Plaintext $\mathbf{m} \in R_p$ On Input A Ciphertext $\mathfrak{c}_{\mathbf{m}} = \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m})$.

Protocol Π_{PREP}

Usage: Note that DataGeneration can be run in four distinct threads, and DataCheck in two threads with one thread executing the Square and Shared bit checking at the same time. Each thread executes its own check for correct broadcasting using Section 3.1.

Initialize: This produces the keys for encryption and MACs. On input (Start, p) from all the players:

1. The players call $\mathcal{F}_{\text{SHE}}$ on query KeyGen so player i obtains $(\mathfrak{s}_i, \mathfrak{pk}, \mathfrak{enc})$.
2. The players call $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(\mathbb{F}_p)$ so player j obtains a share α_j of the MAC key, and all players get $\mathfrak{c}_i$, and encryption of α_i, for $1 \leq i \leq n$.
3. All players set $\mathfrak{c}_\alpha \leftarrow \mathfrak{c}_1 + \cdots + \mathfrak{c}_n$.

Data Generation: On input $(\mathsf{DataGen}, n_I, n_m, n_s, n_b)$, the players execute the following subprocedures of DataGen from Figure 13 and Figure 14:

1. $\mathsf{InputProduction}(n_I)$
2. $\mathsf{Triples}(n_m)$
3. $\mathsf{Squares}(n_s)$
4. $\mathsf{Bits}(n_b)$

Data Check: On input DataCheck, the players do the following:

1. Generate two random values t_m, t_{sb} running the steps below twice:
   (a) Every player P_i samples random $t_i \leftarrow \mathbb{F}_p$ and asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_i^t \leftarrow \mathsf{Commit}(t_i)$.
   (b) Every player P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^t)$ and all players obtain t_j for $1 \leq j \leq n$.
   (c) Every player sets $t \leftarrow t_1 + \cdots + t_n$. If $t = 0$, then repeat the previous steps.
2. Execute $\mathsf{DataCheck}(t_m, t_{sb})$.

Finalize: For the set of partially opened values run protocol MACCheck from Figure 10.

Abort: If $\mathcal{F}_{\text{SHE}}$ outputs a set S of corrupted players at any time, all players output S, and the protocol aborts.

Fig. 12. The Preprocessing Phase

Procedure DataGen

Input Production: This produces at least $n_I \cdot n$ shared values $r_{i,j}$ for $1 \leq i \leq n_I$ and $1 \leq j \leq n$ such that player j holds the actual value $r_{i,j}$ and all other players hold a sharing of this value only.

1. For $j \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, \lceil 2 \cdot n_I/m \rceil\}$.
   (a) Player j generates $\mathbf{r} \in R_p$.
   (b) Player j computes $\mathfrak{c} \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{r})$ and broadcasts the ciphertext to all players.
   (c) The parties execute $\mathsf{Reshare}(\mathfrak{c}, \mathsf{NoNewCiphertext})$ so that player i obtains the share $\mathbf{r}_i$ of $\mathbf{r}$
   (d) All parties compute $\mathfrak{c}_{\gamma(\mathbf{r})} \leftarrow \mathfrak{c}_{\mathbf{r}} \cdot \mathfrak{c}_{\alpha}$.
   (e) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{r})}, \mathsf{NoNewCiphertext})$. to obtain shares $\gamma(\mathbf{r})_i$.
   (f) Player i decomposes the plaintext elements $\mathbf{r}_i$ and $\gamma(\mathbf{r})_i$ into their $m/2$ slot values via the FFT and locally stores the resulting data.
   (g) Player j does the same with $\mathbf{r}$ to obtain the values $r_{(k-1)\cdot m/2+i,j}$ for $i = 1, \ldots, m/2$.

Triples: This produces at least $2 \cdot n_m$ $\langle\cdot\rangle$-shared values (a_j, b_j, c_j) such that $c_j = a_j \cdot b_j$.

1. For $k \in \{1, \ldots, \lceil 4 \cdot n_m/m \rceil\}$.
   (a) The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that player i obtains plaintext $\mathbf{a}_i$ and all players obtain $\mathfrak{c}_{\mathbf{a}_i}$ an encryption of $\mathbf{a}_i$.
   (b) The players compute $\mathfrak{c}_{\mathbf{a}} \leftarrow \mathfrak{c}_{\mathbf{a}_1} + \cdots + \mathfrak{c}_{\mathbf{a}_n}$ We define $\mathbf{a} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$, although no party can compute $\mathbf{a}$.
   (c) The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that player i obtains plaintext $\mathbf{b}_i$ and all players obtain $\mathfrak{c}_{\mathbf{b}_i}$ an encryption of $\mathbf{b}_i$.
   (d) The players compute $\mathfrak{c}_{\mathbf{b}} \leftarrow \mathfrak{c}_{\mathbf{b}_1} + \cdots + \mathfrak{c}_{\mathbf{b}_n}$ We define $\mathbf{b} = \mathbf{b}_1 + \cdots + \mathbf{b}_n$, although no party can compute $\mathbf{b}$.
   (e) All parties compute $\mathfrak{c}_{\mathbf{a}\cdot\mathbf{b}} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\mathbf{b}}$.
   (f) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\mathbf{a}\cdot\mathbf{b}}, \mathsf{NewCiphertext})$ so that player i obtains the share $\mathbf{c}_i$ and all players obtain a ciphertext $\mathfrak{c}_{\mathbf{c}}$ encrypting the plaintext $\mathbf{c} = \mathbf{c}_1 + \cdots + \mathbf{c}_n$.
   (g) All parties compute $\mathfrak{c}_{\gamma(\mathbf{a})} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\alpha}$, $\mathfrak{c}_{\gamma(\mathbf{b})} \leftarrow \mathfrak{c}_{\mathbf{b}} \cdot \mathfrak{c}_{\alpha}$ and $\mathfrak{c}_{\gamma(\mathbf{c})} \leftarrow \mathfrak{c}_{\mathbf{c}} \cdot \mathfrak{c}_{\alpha}$.
   (h) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{a})}, \mathsf{NoNewCiphertext})$, $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{b})}, \mathsf{NoNewCiphertext})$ and $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{c})}, \mathsf{NoNewCiphertext})$ to obtain shares $\gamma(\mathbf{a})_i$, $\gamma(\mathbf{b})_i$ and $\gamma(\mathbf{c})_i$.
   (i) Player i decomposes the various plaintext elements into their $m/2$ slot values via the FFT and locally stores the resulting $m/2$ multiplication triples.

Fig. 13. Production Of Tuples and Shared Bits

Procedure DataGen

Squares: This produces at least $(2 \cdot n_s + n_b)$ $\langle\cdot\rangle$-shared values (a_j, b_j) such that $b_j = a_j \cdot a_j$.

1. For $k \in \{1, \ldots, \lceil 2 \cdot (2 \cdot n_s + n_b)/m \rceil\}$.
   (a) The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that player i obtains plaintext $\mathbf{a}_i$ and all players obtain $\mathfrak{c}_{\mathbf{a}_i}$ an encryption of $\mathbf{a}_i$.
   (b) The players compute $\mathfrak{c}_{\mathbf{a}} \leftarrow \mathfrak{c}_{\mathbf{a}_1} + \cdots + \mathfrak{c}_{\mathbf{a}_n}$ We define $\mathbf{a} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$, although no party can compute $\mathbf{a}$.
   (c) All parties compute $\mathfrak{c}_{\mathbf{a}^2} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\mathbf{a}}$.
   (d) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\mathbf{a}^2}, \mathsf{NewCiphertext})$ so that player i obtains the share $\mathbf{b}_i$ and all players obtain a ciphertext $\mathfrak{c}_{\mathbf{b}}$ encrypting the plaintext $\mathbf{b} = \mathbf{b}_1 + \cdots + \mathbf{b}_n$.
   (e) All parties compute $\mathfrak{c}_{\gamma(\mathbf{a})} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\alpha}$ and $\mathfrak{c}_{\gamma(\mathbf{b})} \leftarrow \mathfrak{c}_{\mathbf{b}} \cdot \mathfrak{c}_{\alpha}$.
   (f) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{a})}, \mathsf{NoNewCiphertext})$ and $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{b})}, \mathsf{NoNewCiphertext})$ to obtain shares $\gamma(\mathbf{a})_i$ and $\gamma(\mathbf{b})_i$.
   (g) Player i decomposes the various plaintext elements into their $m/2$ slot values via the FFT and locally stores the resulting $m/2$ squaring tuples.

Bits: This produces at least n_b $\langle\cdot\rangle$-shared values b_j such that $b_j \in \{0, 1\}$.

1. For $k \in \{1, \ldots, \lceil 2 \cdot n_b/m \rceil + 1\}$.[a]
   (a) The players run $\mathcal{F}_{\text{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that player i obtains plaintext $\mathbf{a}_i$ and all players obtain $\mathfrak{c}_{\mathbf{a}_i}$ an encryption of $\mathbf{a}_i$.
   (b) The players compute $\mathfrak{c}_{\mathbf{a}} \leftarrow \mathfrak{c}_{\mathbf{a}_1} + \cdots + \mathfrak{c}_{\mathbf{a}_n}$ We define $\mathbf{a} = \mathbf{a}_1 + \cdots + \mathbf{a}_n$, although no party can compute $\mathbf{a}$.
   (c) All parties compute $\mathfrak{c}_{\mathbf{a}^2} \leftarrow \mathfrak{c}_{\mathbf{a}} \cdot \mathfrak{c}_{\mathbf{a}}$.
   (d) The players invoke protocol $\mathsf{DistDec}$ to decrypt $\mathfrak{c}_{\mathbf{a}^2}$ and thereby obtain $\mathbf{s} = \mathbf{a}^2$.
   (e) If any slot position in $\mathbf{s}$ is equal to zero then set it to one. .
   (f) A fixed square root $\mathbf{t}$ of $\mathbf{s}$ is taken, say the one for which each slot position is odd when represented in $[1, \ldots, p)$.
   (g) Compute $\mathfrak{c}_{\mathbf{v}} \leftarrow \mathbf{t}^{-1} \cdot \mathfrak{c}_{\mathbf{a}}$, this is an encryption of $\mathbf{v} = \mathbf{t}^{-1} \cdot \mathbf{a}$, which is a message for which each slot position contains $\{-1, 1\}$, bar the one which we replaced in step (1e).
   (h) All parties compute $\mathfrak{c}_{\gamma(\mathbf{v})} \leftarrow \mathfrak{c}_{\mathbf{v}} \cdot \mathfrak{c}_{\alpha}$.
   (i) The parties execute $\mathsf{Reshare}(\mathfrak{c}_{\mathbf{v}}, \mathsf{NoNewCiphertext})$ and $\mathsf{Reshare}(\mathfrak{c}_{\gamma(\mathbf{v})}, \mathsf{NoNewCiphertext})$ to obtain shares $\mathbf{v}_i$ and $\gamma(\mathbf{v})_i$.
   (j) Player i decomposes the various plaintext elements into their slot values via the FFT, bar the ones replaced in step (1e) to obtain $\langle v_j \rangle$ for $j = 1, \ldots, B$ where $B \approx m \cdot (p-1)/(2 \cdot p)$.
   (k) Set $\langle b_j \rangle \leftarrow (1/2) \cdot (\langle v_j \rangle + 1)$ and output $\langle b_j \rangle$.

[a] Notice that in the production of shared bits the number of rounds is one more than one would expect at first glance: this is because some entry of the input vector may be equal to zero, making such entry unusable for the procedure. This event happens with probability $1/p$, so the expected number of bits produced per iteration is $m \cdot (p-1)/(2 \cdot p)$, rather than $m/2$ (if no entry were zero). Therefore, in order to produce at least n_b elements, we add an extra round to the procedure.

Fig. 14. Production Of Tuples and Shared Bits (continued)

Procedure DataCheck

Usage: Note that all players have previously agreed on two common random values t_m, t_{sb}.

Checking Multiplication Triples: This produces at least n_m checked $\langle\cdot\rangle$-shared values (a_j, b_j, c_j) such that $c_j = a_j \cdot b_j$.

1. For $k \in \{1, \ldots, n_m\}$.
   (a) Take two unused multiplication tuples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, $(\langle f\rangle, \langle g\rangle, \langle h\rangle)$ from the list determined earlier.
   (b) Partially open $t_m \cdot \langle a\rangle - \langle f\rangle$ to obtain ρ and $\langle b\rangle - \langle g\rangle$ to obtain σ.
   (c) Evaluate $t_m \cdot \langle c\rangle - \langle h\rangle - \sigma \cdot \langle f\rangle - \rho \cdot \langle g\rangle - \sigma \cdot \rho$ and partially open the result to obtain τ.
   (d) If $\tau \neq 0$ then output $\varnothing$ and abort.
   (e) Output $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ as a valid multiplication triple.

Checking Squaring Tuples: This produces at least n_s checked $\langle\cdot\rangle$-shared values (a_j, b_j) such that $b_j = a_j^2$.

1. For $k \in \{1, \ldots, n_s\}$.
   (a) Take two unused squaring tuples $(\langle a\rangle, \langle b\rangle)$, $(\langle f\rangle, \langle h\rangle)$ from the list determined earlier.
   (b) Partially open $t_{sb} \cdot \langle a\rangle - \langle f\rangle$ to obtain ρ.
   (c) Evaluate $t_{sb}^2 \cdot \langle b\rangle - \langle h\rangle - \rho \cdot (t_{sb} \cdot \langle a\rangle + \langle f\rangle)$ and partially open the result to obtain τ.
   (d) If $\tau \neq 0$ then output $\varnothing$ and abort.
   (e) Output $(\langle a\rangle, \langle b\rangle)$ as a valid squaring tuple.

Checking Shared Bits: This produces at least n_b checked $\langle\cdot\rangle$-shared values b_j such that $b_j \in \{0, 1\}$.

1. For $k \in \{1, \ldots, n_b\}$.
   (a) Take an unused squaring tuples $(\langle f\rangle, \langle h\rangle)$ and an unused bit sharing $\langle a\rangle$ from the lists determined earlier.
   (b) Partially open $t_{sb} \cdot \langle a\rangle - \langle f\rangle$ to obtain ρ.
   (c) Evaluate $t_{sb}^2 \cdot \langle a\rangle - \langle h\rangle - \rho \cdot (t_{sb} \cdot \langle a\rangle + \langle f\rangle)$ and partially open the result to obtain τ.
   (d) If $\tau \neq 0$ then output $\varnothing$ and abort.
   (e) Output $\langle a\rangle$ as a valid bit sharing.

Fig. 15. Check The Output Of The Data Production Procedure

D.2 Functionalities

The functionality $\mathcal{F}_{\text{PREP}}$

Let A be the set of indices of corrupted players. Symbols in bold denote vectors in $(\mathbb{F}_p)^k$. Arithmetic is componentwise.

Initialize: On input (Start, p) from honest players and adversary, the functionality sets the internal flag BreakDown to false and then it does the following:

1. For each corrupted player $i \in A$, the functionality accepts shares α_i from the adversary, and it samples at random α_i for each $i \notin A$. Then the functionality sets $\alpha \leftarrow \alpha_1 + \cdots + \alpha_n$.
2. The functionality waits for signal Abort, Proceed or Cheat from the adversary.
3. If received Proceed, the functionality outputs α_i to player i.
4. Otherwise, and if the functionality did not abort in Cheat, it outputs adversary's contribution Δ_i to player i.

Computation: On input DataGen from all honest players and adversary, and only if the functionality received Proceed (or BreakDown is true) it executes the data generation procedures specified in Figure 17.

Macro $\mathsf{Angle}(\mathbf{v}_1, \ldots, \mathbf{v}_n, \Delta_\gamma, k)$ The above will be run by the functionality at several points to create representations $\langle \cdot \rangle$.

1. It gets $\{\gamma_i\}_{i \in A}$ from the adversary.
2. Let $\mathbf{v} = \mathbf{v}_1 + \cdots + \mathbf{v}_n$, set $\gamma(\mathbf{v}) \leftarrow \alpha \cdot \mathbf{v} + \Delta_\gamma$.
3. Sample at random $\gamma_i(\mathbf{v}) \leftarrow (\mathbb{F}_p)^k$ for $i \notin A$, subject to $\gamma(\mathbf{v}) = \sum_i^n \gamma(\mathbf{v})_i$.
4. Return $(\gamma(\mathbf{v})_1, \ldots, \gamma(\mathbf{v})_n)$.

Cheat: The functionality chooses to do either one of the following:

- It sends, with probability $1/c$, Success to the adversary and sets the internal flag BreakDown to true.
- Otherwise it sends NoSuccess to the adversary and players, and goes to "Abort".

Abort: The functionality waits for $S \subseteq A$ from the adversary and then outputs S to all players.

Fig. 16. MAC Generation and Covert Procedures to Generate Auxiliar Data

The functionality $\mathcal{F}_{\text{PREP}}$ (continued)

Let A be the set of indices of corrupted players. Symbols in bold denote vectors in $(\mathbb{F}_p)^k$. Arithmetic is componentwise.

Input Production: On input $\mathsf{DataType} = (\mathsf{InputPrep}, n_I)$,

1. The functionality choose random values $I = \{\mathbf{r}^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \notin A\}$.
2. It accepts from the adversary corrupted values $\{\mathbf{r}^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \in A\}$, corrupted shares $\{\mathbf{r}_k^{(i)} \in (\mathbb{F}_p)^{n_I} \mid k \in A, i \leq n\}$, and offset for data and MACs $\{\Delta_r^{(i)}, \Delta_\gamma^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \leq n\}$. Then it does the following:
   (a) Sample honest shares $\{\mathbf{r}_k^{(i)} \mid k \notin A, i \leq n\}$ subject to $\mathbf{r}^{(i)} + \Delta_r^{(i)} = \sum_{k=1}^n \mathbf{r}_k^{(i)}$.
   (b) Run macro $\mathsf{Angle}(\mathbf{r}_1^{(i)}, \ldots, \mathbf{r}_n^{(i)}, \Delta_\gamma^{(i)}, n_I)$, for $i \leq n$.
   (c) Output $\{\mathbf{r}^{(i)}, (\mathbf{r}_i^{(j)}, \gamma_i(\mathbf{r}^{(j)}))_{j \leq n}\}$ to player i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to player i.

Multiplication Triples: On input $\mathsf{DataType} = (\mathsf{Triples}, n_m)$,

1. Choose $2 \cdot n_m$ honest shares $I = \{(\mathbf{a}_i, \mathbf{b}_i) \in (\mathbb{F}_p)^{2 \cdot n_m} \mid i \notin A\}$.
2. It accepts corrupted shares $\{(\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i) \in (\mathbb{F}_p)^{3 \cdot n_m} \mid i \in A\}$ and MAC offsets $\{(\Delta_\gamma^{(a)}, \Delta_\gamma^{(b)}, \Delta_\gamma^{(c)}) \in (\mathbb{F}_p)^{3 \cdot n_m}\}$ from the adversary. It performs the following:
   (a) Set $\mathbf{c} \leftarrow (\mathbf{a}_1 + \cdots + \mathbf{a}_n) \cdot (\mathbf{b}_1 + \cdots + \mathbf{b}_n)$.
   (b) Compute a set of honest shares $\{\mathbf{c}_i \mid i \notin A\}$ subject to $\mathbf{c} = \sum_{i=1}^n \mathbf{c}_i$.
   (c) Run the macros $\mathsf{Angle}(\mathbf{a}_1, \ldots, \mathbf{a}_n, \Delta_\gamma^{(a)}, n_m)$, $\mathsf{Angle}(\mathbf{b}_1, \ldots, \mathbf{b}_n, \Delta_\gamma^{(b)}, n_m)$, $\mathsf{Angle}(\mathbf{c}_1, \ldots, \mathbf{c}_n, \Delta_\gamma^{(c)}, n_m)$.
   (d) Output $\{(\mathbf{a}_i, \gamma_i(\mathbf{a})), (\mathbf{b}_i, \gamma_i(\mathbf{b})), (\mathbf{c}_i, \gamma_i(\mathbf{c}))\}$ to player i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to player i.

Squaring Tuples: On input $\mathsf{DataType} = (\mathsf{Squares}, n_s)$,

1. Choose $N = n_s$ honest shares $I = \{\mathbf{a}_i \in (\mathbb{F}_p)^{n_s} \mid i \notin A\}$.
2. It accepts corrupted shares $\{(\mathbf{a}_i, \mathbf{s}_i) \in (\mathbb{F}_p)^{2 \cdot n_s} \mid i \in A\}$ and MAC offsets $\{(\Delta_\gamma^{(a)}, \Delta_\gamma^{(s)}) \in (\mathbb{F}_p)^{2 \cdot n_s}\}$ from the adversary. It does the following:
   (a) Set $\mathbf{s} \leftarrow (\mathbf{a}_1 + \cdots + \mathbf{a}_n) \cdot (\mathbf{a}_1 + \cdots + \mathbf{a}_n)$.
   (b) Compute a a set of honest shares $\{\mathbf{s}_i \mid i \notin A\}$ subject to $\mathbf{s} = \sum_{i=1}^n \mathbf{s}_i$.
   (c) Run the macro $\mathsf{Angle}(\mathbf{a}_1, \ldots, \mathbf{a}_n, \Delta_\gamma^{(a)}, n_s)$ and $\mathsf{Angle}(\mathbf{s}_1, \ldots, \mathbf{s}_n, \Delta_\gamma^{(s)}, n_s)$.
   (d) Output $\{(\mathbf{a}_i, \gamma_i(\mathbf{a})), (\mathbf{s}_i, \gamma_i(\mathbf{s}))\}$ to player i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to player i.

Shared Bits: On input $\mathsf{DataType} = (\mathsf{Bits}, n_b)$,

1. It gets shares $\{\mathbf{b}_i \in (\mathbb{F}_p)^{n_b} \mid i \in A\}$ and MAC offsets $\{\Delta_\gamma^{(b)} \in (\mathbb{F}_p)^{n_b}\}$ from the adversary.
   (a) Uniformly sample n_b honest shares $I = \{\mathbf{b}_i \in (\mathbb{F}_p)^{n_b} \mid i \notin A\}$ subject to the condition $\sum_i \mathbf{b}_i \in \{0,1\}^{n_b}$.
   (b) Run the macro $\mathsf{Angle}(\mathbf{b}_1, \ldots, \mathbf{b}_n, \Delta_\gamma^{(b)}, n_b)$.
   (c) Output $(\mathbf{b}_i, \gamma_i(\mathbf{b}))$ to player i, or if $\mathsf{BreakDown}$ is true, output adversary's contribution Δ_i to player i.

Fig. 17. Operations to Generate Auxiliar Data for the Online Phase

D.3 Proof of Lemma 1

Proof.

We here inspect the correctness and the soundness error of the MACCheck protocol. In order to understand the probability of an adversary being able to cheat, we design the following security game.

1. The challenger generates the secret key $\alpha \leftarrow \alpha_1 + \cdots + \alpha_n$ and MACs $\gamma(a_j)_i \leftarrow \alpha \cdot a_j$ and sends messages $a_1, \ldots, a_t$ to the adversary.
2. The adversary sends back messages $a'_1, \ldots, a'_t$.
3. The challenger generates random values $r_1, \ldots, r_t \leftarrow \mathbb{F}_p$ and sends them to the adversary.
4. The adversary provides an error Δ.
5. Set $a \leftarrow \sum_{j=0}^{t} r_j a'_j$, $\gamma_i \leftarrow \sum_{j=0}^{t} r_j \gamma(a_j)_i$, and $\sigma_i \leftarrow \gamma_i - \alpha_i \cdot a$. Now, the challenger checks that $\sigma_1 + \cdots + \sigma_n = \Delta$

The adversary wins the game if there is an i for which $a'_i \neq a_i$ and the final check goes through.

The second step in the game where the adversary sends the a'_i's models the fact that corrupted players can choose to lie about their shares of values opened during the protocol execution. Δ models the fact that the adversary is allowed to introduce errors on the macs.

Now, let us look at the probability of winning the game if the r_i's are randomly chosen. If the check goes through, we have that the following equalities hold:

$$\begin{aligned}
\Delta &= \sum_{i=1}^{n} \sigma_i = \sum_{i=1}^{n} (\gamma_i - \alpha_i \cdot a) \\
&= \sum_{i=1}^{n} \left(\sum_{j=1}^{t} r_j \cdot \gamma(a_j)_i - \alpha_i \cdot \sum_{j=1}^{t} r_j \cdot a'_j \right) \\
&= \sum_{i=1}^{n} \left(\sum_{j=1}^{t} \left(r_j \cdot \gamma(a_j)_i - \alpha_i \cdot r_j \cdot a'_j \right) \right) \\
&= \sum_{j=1}^{t} \left(r_j \cdot \sum_{i=1}^{n} \left(\gamma(a_j)_i - \alpha_i \cdot a'_j \right) \right) \\
&= \sum_{j=1}^{t} r_j \cdot \left(\alpha \cdot a_j - \alpha \cdot a'_j \right) \\
&= \alpha \cdot \sum_{j=1}^{t} r_j \cdot \left(a_j - a'_j \right)
\end{aligned}$$

So, the following equality holds:

$$\alpha \cdot \sum_{j=0}^{t} r_j (a'_j - a_j) = \Delta. \tag{1}$$

First we consider the case where $\sum_{j=0}^{t} r_j (a'_j - a_j) \neq 0$, so $\alpha = \Delta / \sum_{j=0}^{t} r_j (a'_j - a_j)$. This implies that being able to pass the check is equivalent to guessing α. However, since the adversary has no information about α, this happens with probability only $1/|\mathbb{F}_p|$. So what is left is to argue that $\sum_{j=0}^{t} r_j (a'_j - a_j) = 0$ also happens with very low probability. This can be seen as follows. We define $\mu_j := (a'_j - a_j)$ and $\mu := (\mu_1, \ldots, \mu_t), r := (r_1, \ldots, r_t)$. Now $f_\mu(r) := r \cdot \mu = \sum_{j=0}^{t} r_j \mu_j$ defines a linear mapping, which is not the 0-mapping since at least one $\mu_j \neq 0$. From linear algebra we then have the rank-nullity theorem telling us that $\dim(\ker(f_\mu)) = t - 1$. Also since r is random and the adversary does not know r when choosing the a'_i's, the probability of $r \in \ker(f_\mu)$ is $|\mathbb{F}_p^{t-1}|/|\mathbb{F}_p^t| = 1/|\mathbb{F}_p|$. Summing up, the total probability of winning the game is at most $2/|\mathbb{F}_p|$.

For correctness we use the fact that Equation 1 holds with probability one if $a'_j = a_j$ and $\Delta = 0$ (honest prover).

□

D.4 Proof of Theorem 4

Proof. We construct a simulator $\mathcal{S}_{\text{PREP}}$ (given in Figure 18 and Figure 19) such that no polynomial-time environment can distinguish, with significant probability, a view obtained running Π_{PREP} from a view obtained running $\mathcal{S}_{\text{PREP}} \diamond \mathcal{F}_{\text{PREP}}$. The environment's view is the collection of all intermediate messages that corrupted players send and receive, plus the inputs and outputs of all players.

In a nutshell, the simulator will run a copy of Π_{PREP} with the adversary, acting on behalf of honest players. Keys for the underlying cryptosystem and MACs are generated by simulating queries KeyGen and EncCommit to $\mathcal{F}_{\text{SHE}}$ respectively. Note that due to the distributed decryption, data for the (online) input preparation stage might be incorrectly secret shared, and all type of data might be incorrectly MAC'd. Since the simulator knows α and $\mathfrak{s}$, it can compute offsets on the secret sharing and MACs and pass them to $\mathcal{F}_{\text{PREP}}$.

Before we discuss indistinguishability we explain how the cheat mechanism is handled in the simulation. In the execution of Π_{PREP}, the environment may send Cheat either in the initial query KeyGen or in any later query EncCommit to $\mathcal{F}_{\text{SHE}}$. Thus, the success probability depends on the number of cheat attempts. The simulator ensures two things: 1) Whenever the environment sends the *first* Cheat to what it thinks is $\mathcal{F}_{\text{SHE}}$, the call is forwarded to $\mathcal{F}_{\text{PREP}}$, which decides whether or not it is successful. 2) Assuming this cheat was successful, the simulator recreates the success probability that a real interaction would have. This is needed as otherwise the environment would distinguish. The inner procedure SEncCommit is designed for this purpose.

We now turn to show indistinguishability. We point out that there is mainly one difference between a simulated run and a real execution of Π_{PREP}: In a simulated run, honest shares used in the interaction are randomly sampled by the simulator. These shares correspond to the MAC key, and shares of generated data together with the shares of their MACs. At the end of the day, $\mathcal{F}_{\text{PREP}}$ will output data using its own honest shares of α, and its own honest shares of data and MACs.

We can split the view of the environment in four chunks. Namely, messages interchanged either in DataGen, in DataCheck, or in MACCheck, and players' output of $\mathcal{F}_{\text{PREP}}$. Clearly, indistinguishability of simulated and real views of DataGen chunk comes from the semantic property of the underlying cryptosystem. For the DataCheck chunk, note that all opened values are a combination of output data and sacrificed data. The latter does not form part of the final output, and therefore by no means the environment can reconstruct the set of opened values using its view, as it does not know honest shares of the sacrificed data. In other words, openings are randomized via sacrificings from the environment's point of view, so the best it can do is to guess sacrificed honest shares, which happens with probability $1/|\mathbb{F}_p|$ for each share's guessing. For the MACCheck chunk, we refer to the fact that the soundness error of MACCheck is $2/p$, as shown in Lemma 1. Both probabilities are negligible if p is exponential in the security parameter. Lastly, we also have consistency between the output of $\mathcal{F}_{\text{PREP}}$ and what the environment sees in corrupted transcripts. This is due to the fact that the offsets (those quantities denoted by Δ) are simply the difference between deviated and correctly computed data, and therefore independent of what data refers to.

If the protocol aborts in DataCheck or MACCheck, the players output $\varnothing$, and so does $\mathcal{F}_{\text{PREP}}$ on instruction of the simulator. This corresponds to the fact that those protocols do not reveal the identity of any corrupted party.

It remains to show what happens in case Cheat or Abort is sent by the environment. If the cheat did not go through, players' output is a single message S for a set S of corrupted players in both real and simulated interaction. On the other hand, if the cheat did go through, the functionality $\mathcal{F}_{\text{PREP}}$ breaks down, and the simulator can decide what MAC key is used and what data is outputted to every player, so it just gives to $\mathcal{F}_{\text{PREP}}$ what it has been generated during the interaction. If the environment sends Abort and a set S of corrupted players, this is simply passed to $\mathcal{F}_{\text{PREP}}$, which forwards it to the players.

□

The simulator $\mathcal{S}_{\text{PREP}}$

Initialize:

- The simulator first sends (Start, p) to $\mathcal{F}_{\text{PREP}}$ and then interacts with the adversary acting as $\mathcal{F}_{\text{SHE}}$ on query KeyGen to generate the encryption public key $(\mathfrak{pk}, \mathfrak{enc})$ and a complete set of shares $\{\mathfrak{s}_1, \ldots, \mathfrak{s}_n\}$ of the secret key. If the adversary sends Cheat to $\mathcal{F}_{\text{SHE}}$, the simulator forwards it to $\mathcal{F}_{\text{PREP}}$. If the cheat passed through, the simulator sets the flag BreakDown to true, otherwise it is set to false.
- The generation of the MAC key α is done as in the protocol, but calling to $\mathsf{SEncCommit}(\mathbb{F}_p)$ instead to $\mathcal{F}_{\text{SHE}}$ on query EncCommit. The simulator stores $\alpha \leftarrow \alpha_1 + \cdots + \alpha_n$ for later use.
- Lastly, it gives α_i to $\mathcal{F}_{\text{PREP}}$ for $i \in A$ if BreakDown is false, and $i \leq n$ otherwise.
- If the simulation $\mathcal{F}_{\text{SHE}}$ aborts on KeyGen or and EncCommit, go to "Abort".

Command = DataGen: On input (n_I, n_m, n_s, n_b) from honest players and adversary, the simulator sets

$\mathcal{T}_{Input} \leftarrow \mathsf{SimDataGen}(\mathsf{InputPrep}, n_I)$
$\mathcal{T}_{Triples} \leftarrow \mathsf{SimDataGen}(\mathsf{Triples}, n_m)$
$\mathcal{T}_{Squares} \leftarrow \mathsf{SimDataGen}(\mathsf{Squares}, n_s)$
$\mathcal{T}_{Bits} \leftarrow \mathsf{SimDataGen}(\mathsf{Bits}, n_b)$,

where SimDataGen is specified in Figure 19. These calls also return a decision bit. If it is set to Abort, the simulator goes to "Abort".

Command = DataCheck:

- Step 1 is executed as in the protocol but calling to $\mathsf{SEncCommit}(R_p)$. The simulator goes to "Abort" if SEncCommit says so.
- The simulator performs steps (a)-(d) of subprocedures Triples, Squares, Bits of DataCheck. In each iteration k, it gets to know the value σ_k. If any of these values are non-zero, the simulator sends Abort and $\varnothing$ to $\mathcal{F}_{\text{PREP}}$. Otherwise, the algebraic relation among generated data is correct with probability $1 - 1/p$.

Finalize: At this point, the functionality is waiting for instruction Proceed or Abort, or otherwise, a complete break down occurred, and the functionality is waiting for command DataGen and output values from the adversary.

1. The simulator engages with the adversary in a normal run of MACCheck on behalf of each honest player i. Note that to generate honest σ_i the simulator uses shares α_i. If $\sigma_1 + \cdots + \sigma_n \neq 0$, send Abort and $\varnothing$ to $\mathcal{F}_{\text{PREP}}$.
2. Otherwise send Success to the adversary, and send to $\mathcal{F}_{\text{PREP}}$ the following:
   - If BreakDown is false, send $\mathcal{T}_{Input}$, $\mathcal{T}_{Triples}$, $\mathcal{T}_{Squares}$, $\mathcal{T}_{Bits}$.
   - If BreakDown is true, send all the data (corresponding to honest and corrupted players) generated in the execution of SimDataGen.

Abort: If the simulated $\mathcal{F}_{\text{SHE}}$ aborts outputting a set S of corrupted players, input Abort and S to $\mathcal{F}_{\text{PREP}}$.

Fig. 18. The Simulator $\mathcal{S}_{\text{PREP}}$ For The Preprocessing Phase

The simulator $\mathcal{S}_{\text{PREP}}$

SimDataGen(DataType): This procedure gets ready the data to be inputted to $\mathcal{F}_{\text{PREP}}$.

DataType $=$ InputPrep :

- The simulator engages in a normal run of steps (a)-(g) calling to SReshare instead of Reshare. If, at any point, some of the calls returned Abort, the simulator sets Decision $\leftarrow$ Abort and $\mathcal{T}_{Input} \leftarrow \varnothing$.
- Otherwise all the rounds were successful. The simulator sets Decision $\leftarrow$ Continue. Note that in step (c) (after unpacking all the rounds), the simulator gets players' shares and MAC shares $\{\hat{\mathbf{r}_k}^{(i)}, \gamma_k^{(i)} \in (\mathbb{F}_p)^{2 \cdot n_I} \mid i, k \leq n\}$. Then $\hat{\mathbf{r}}^{(i)} = \sum_k^n \hat{\mathbf{r}}_k^{(i)}$ is the (presumably) input of player i. The simulator has the secret key, so it can get the real input $\mathbf{r}^{(i)}$ from the broadcast ciphertexts (if P_i is corrupted) or from what he generated (if P_i is honest). It computes offsets $\Delta_r^{(i)} \leftarrow \hat{\mathbf{r}}^{(i)} - \mathbf{r}^{(i)}$ and $\Delta_\gamma^{(i)} \leftarrow \sum_k^n \gamma_k^{(i)} - \alpha \cdot \mathbf{r}^{(i)}$
  There are two possibilities:
  - Flag BreakDown is set to false. This means no cheat has occurred, so the simulator prepares corrupt inputs, corrupt shares and MAC shares, and offsets. That is, it sets $\mathcal{T}_{Input} \leftarrow \{\mathbf{r}^{(k)}, \hat{\mathbf{r}}_k^{(i)}, \Delta_r^{(i)}, \Delta_\gamma^{(i)}, \gamma_k^{(i)} \mid k \in A, i \leq n\}$
  - Flag BreakDown is set to true. Then there was at least one successful cheat, and the functionality is waiting for adversary's contributions. The simulator sets $\mathcal{T}_{Input}$ to be the output of each player.

DataType $=$ Triples, Squares, Bits: The simulator engages in a normal run of the subprocedure specified by DataType, but calling to SEncCommit(R_p) and SReshare$(\mathfrak{c}_{\mathbf{m}})$ instead of $\mathcal{F}_{\text{ENCCOMMIT}}$ and Reshare$(\mathfrak{c}_{\mathbf{m}})$. If any of the above macros returned Abort the simulator sets Decision $\leftarrow$ Abort and $\mathcal{T}_{DataType} \leftarrow \varnothing$. In any other case the simulator sets Decision $\leftarrow$ Continue, handles the BreakDown flag as above, and does:

Triples: Set $\mathcal{T}_{Triples} \leftarrow \{(\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i, \gamma(\mathbf{a})_i, \gamma(\mathbf{b})_i, \gamma(\mathbf{c})_i, \Delta_\gamma^{(a)}, \Delta_\gamma^{(b)}, \Delta_\gamma^{(c)}) \in (\mathbb{F}_p)^{9 \cdot (2 \cdot n_m)} \mid i \leq n\}$. The shares are unpacked in step (i): corrupt shares are given by the adversary, and honest shares are sampled uniformly. MAC shares are produced after executing SReshare to simulate step (h), and the offsets are computed as explained earlier.

Squares: Set $\mathcal{T}_{Squares} \leftarrow \{(\mathbf{a}_i, \mathbf{b}_i, \gamma(\mathbf{a})_i, \gamma(\mathbf{b})_i, \Delta_\gamma^{(a)}, \Delta_\gamma^{(b)}) \in (\mathbb{F}_p)^{6 \cdot (2 \cdot n_s + n_b)} \mid i \leq n\}$ Shares, MAC shares and offsets are obtained as explained above.

Bits: Set $\mathcal{T}_{Bits} \leftarrow \{(\mathbf{b}_i, \gamma_i, \Delta_\gamma^{(b)}) \in (\mathbb{F}_p)^{3 \cdot (2 \cdot n_b')} \mid i \leq n\}$. A number $n_b' \geq n_b$ of binary shares and MACs has been computed, The exact amount n_b' is round-dependent and it is expected to be approximately $(n_b + m/2) \cdot (p-1)/p$.

Return (Decision, $\mathcal{T}_{DataType}$).

Macro SEncCommit(cond) This macro is intended to simulate a call to $\mathcal{F}_{\text{SHE}}$ on query EncCommit.

- The simulator receives corrupted seeds s_i from the adversary, when it thinks is interacting with $\mathcal{F}_{\text{SHE}}$, and computes $\mathbf{m}_i$ and $\mathfrak{c}_{\mathbf{m}_i}$ for $i \in A$ which are given to the adversary. Then the simulator generates uniformly $\mathbf{m}_i$ and $\mathfrak{c}_i = \text{Enc}_{\mathfrak{pk}}(\mathbf{m}_i)$ for $i \notin A$, and gives $\mathfrak{c}_i$ to the adversary. It waits for response Proceed, Cheat or Abort.
- If the adversary gives Proceed, the simulator sets Decision $\leftarrow$ Continue, and if the adversary gives Abort, set Decision $\leftarrow$ Abort and also send Abort to $\mathcal{F}_{\text{PREP}}$.
- If the adversary gives (Cheat, $\{\mathbf{m}_i^*, \mathfrak{c}_i^*\}_{i \in A}$), set $\mathbf{m}_i \leftarrow \mathbf{m}_i^*$, $\mathfrak{c}_i \leftarrow \mathfrak{c}_i^*$ for $i \in A$, and do the following:
  1. Check if flag BreakDown is false, if so, send Cheat to $\mathcal{F}_{\text{PREP}}$. Then set BreakDown to true. There are two possibilities:
     (a) The functionality returns Success: set Decision $\leftarrow$ Continue.
     (b) The functionality returns NoSuccess: set Decision $\leftarrow$ Abort.
  2. If BreakDown is set to true, with probability $1/c$ set Decision $\leftarrow$ Continue, or otherwise Decision $\leftarrow$ Abort.
- Return (Decision, $\mathbf{m}_1, \ldots, \mathbf{m}_n, \mathfrak{c}_1, \ldots, \mathfrak{c}_n$).

Macro SReshare($\mathfrak{c}_{\mathbf{m}}$)

- Set $(\mathbf{f}_1, \ldots, \mathbf{f}_n, \mathfrak{c}_1, \ldots, \mathfrak{c}_n) \leftarrow$ SEncCommit(R_p) and $\mathbf{f} \leftarrow \sum_i \mathbf{f}_i$. Set Decision $\leftarrow$ Abort if SEncCommit says so.
- Otherwise, set Decision $\leftarrow$ Continue and run steps 2-5 of Reshare. Note that in step 3 the simulator might get an invalid value $(\mathbf{m} + \mathbf{f})^*$. Set $\mathbf{m}_1 \leftarrow (\mathbf{m} + \mathbf{f})^* - \mathbf{f}_1$ and $\mathbf{m}_i \leftarrow -\mathbf{f}_i$.
- Return shares (Decision, $\mathbf{m}_1, \ldots, \mathbf{m}_n$).

Fig. 19. Internal Procedures Of The Simulator $\mathcal{S}_{\text{PREP}}$

E Online Phase : Protocol, Functionalities and Simulators

E.1 Protocols

Protocol Π_{ONLINE}

Initialize: The parties call $\mathcal{F}_{\text{PREP}}$ to get the shares α_i of the MAC key, a number of multiplication triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, squares $(\langle a\rangle, \langle b\rangle)$, bits $\langle b\rangle$, and mask values $(r_i, \langle r_i\rangle)$ as needed for the circuit being evaluated. If $\mathcal{F}_{\text{PREP}}$ aborts outputting a set S of corrupted players, the players output S and abort. Then the operations specified below are performed according to the circuit.

Input: To share his input x_i, Player i takes an available mask value $(r_i, \langle r_i\rangle)$ and does the following:

1. Broadcast $\epsilon \leftarrow x_i - r_i$.
2. The players compute $\langle x_i\rangle \leftarrow \langle r_i\rangle + \epsilon$.

Add: On input $(\langle x\rangle, \langle y\rangle)$, the players locally compute $\langle x + y\rangle \leftarrow \langle x\rangle + \langle y\rangle$.

Multiply: On input $(\langle x\rangle, \langle y\rangle)$, the players do the following:

1. Take one multiplication triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ and open $\langle x\rangle - \langle a\rangle, \langle y\rangle - \langle b\rangle$ to get ϵ, ρ respectively.
2. Locally each player sets $\langle z\rangle \leftarrow \langle c\rangle + \epsilon \cdot \langle b\rangle + \rho \cdot \langle a\rangle + \epsilon \cdot \rho$

Square: On input $\langle x\rangle$ the players do the following:

1. Take a square pair $(\langle a\rangle, \langle b\rangle)$ and partially open $\langle x\rangle - \langle a\rangle$ so all players get ϵ.
2. All players locally compute $\langle z\rangle \leftarrow \langle b\rangle + 2 \cdot \epsilon \cdot \langle x\rangle - \epsilon^2$.

Output: This procedure is entered once the players have finished the circuit evaluation, but still the final output $\langle y\rangle$ has not been opened.

1. The players call the MACCheck protocol on input all opened values so far. If it fails, they output $\varnothing$ and abort. $\varnothing$ represents the fact to the corrupted players remain undetected in this case.
2. The players open $\langle y\rangle$ and call MACCheck on input y to verify its MAC. If the check fails, they output $\varnothing$ and abort, otherwise they accept y as a valid output.

Fig. 20. Operations for Secure Function Evaluation

E.2 Functionalities

Functionality $\mathcal{F}_{\text{ONLINE}}$

Initialize: On input $(init, p, k)$ from all parties, the functionality stores $(domain, p, k)$ and waits for an input from the environment. Depending on this, the functionality does the following:

- Proceed It sets BreakDown to false and continues.
- Cheat With probability $1/c$, it sets BreakDown to true, outputs Success to the environment and continues. Otherwise it outputs NoSuccess and proceeds as in Abort.
- Abort It waits for the environment to input a set S of corrupted players, outputs it to the players, and aborts.

Input: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with $varid$ a fresh identifier, the functionality stores $(varid, x)$. If BreakDown is true, it also outputs x to the environment.

Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x + y)$.

Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x \cdot y)$.

Square: On input $(square, varid_1, varid_2)$ from all parties (if $varid_1$ is present in memory and $varid_2$ is not), the functionality retrieves $(varid_1, x)$, and stores $(varid_2, x^2)$.

Output: On input $(output, varid)$ from all honest parties (if $varid$ is present in memory), the functionality retrieves $(varid, y)$ and outputs it to the environment.

- If BreakDown is false, the functionality waits for an input from the environment. If this input is Deliver then y is output to all players. Otherwise $\varnothing$ is output to all players.
- If BreakDown is true, the functionality waits for y^* from the environment and outputs it to all players.

Fig. 21. The ideal functionality for MPC

E.3 Proof of Theorem 5

Proof.

We construct a simulator $\mathcal{S}_{\text{ONLINE}}$ to work on top of the ideal functionality $\mathcal{F}_{\text{ONLINE}}$, such that the adversary cannot distinguish whether it is playing with the protocol Π_{ONLINE} and $\mathcal{F}_{\text{PREP}}$, or the simulator and $\mathcal{F}_{\text{ONLINE}}$. See Appendix E for the complete description of the simulator.

We now proceed with the analysis of the simulation, by first arguing that all the steps before the output are perfectly simulated and finally we show that the simulated output is statistically close to the one in the protocol.

During initialization, the simulator merely acts as $\mathcal{F}_{\text{PREP}}$ with the difference that the decision about the success of a cheating attempt is made by $\mathcal{F}_{\text{ONLINE}}$. If the cheating was successful, $\mathcal{F}_{\text{ONLINE}}$ will output all honest inputs, and the simulator can determine all outputs. Therefore, the simulation will precisely agree with the protocol. For the rest of the proof, we will assume that there was no cheating attempt.

In the input stage the values broadcast by the honest players are uniform in the protocol as well as in the simulation. Addition does not involve communication, while multiplication and squaring involve partial openings: in the protocol a partial opening reveals uniform values, and the same happens also in a simulated run. Moreover, MACs carry the same distribution in both the protocol and the simulation.

In the output stage of both the real and simulated run if the output y is delivered, the environment sees y and the honest players' shares, which are uniform and compatible with y and its MAC. Moreover, in a simulated run the output y is a correct evaluation of the function on the inputs provided by the players in the input phase. In order to conclude, we need to make sure that the same applies to the real protocol with overwhelming probability. As shown in Lemma 1, the adversary was able to cheat in one MACCheck call with probability $2/p$. Thus, the overall cheating probability is negligible since p is assumed to be exponential in the security parameter. This concludes the proof.

□

Simulator $\mathcal{S}_{\text{ONLINE}}$

Initialize: The simulation of the initialization procedure is performed running a local copy of $\mathcal{F}_{\text{PREP}}$. Notice that all the data given to the adversary is know by the simulator.
If the environment inputs Proceed, Cheat, or Abort to the copy of $\mathcal{F}_{\text{PREP}}$, the simulator does so to $\mathcal{F}_{\text{ONLINE}}$ and forwards the output of $\mathcal{F}_{\text{ONLINE}}$ to the environment. If the output is Success, the simulator sets BreakDown to true and uses the environment's inputs as preprocessed data. If $\mathcal{F}_{\text{ONLINE}}$ outputs NoSuccess of the input was Abort, the simulator waits for input S from the environment, forwards it to $\mathcal{F}_{\text{ONLINE}}$, and aborts.

Input:

- If BreakDown is false, honest input is performed according to the protocol, with a dummy input, for example zero.
- If BreakDown is true, $\mathcal{F}_{\text{ONLINE}}$ outputs the inputs of honest players, which then can be used in the simulation.

For inputs given by a corrupt player P_i, the simulator waits for P_i to broadcast the (possibly incorrect) value ϵ', computes $x_i' \leftarrow r_i + \epsilon'$ and uses x_i' as input to $\mathcal{F}_{\text{ONLINE}}$.

Add/Multiply/Square: These procedures are performed according to the protocol. The simulator also calls the respective procedure to $\mathcal{F}_{\text{ONLINE}}$.

Output: $\mathcal{F}_{\text{ONLINE}}$ outputs y to the simulator.

- If BreakDown is false, the simulator now has to provide the honest players' shares of such a value; it already computed an output value y', using the dummy inputs for the honest players, so it can select a random honest player and modify its share adding $y - y'$ and modify the MAC adding $\alpha(y - y')$, which is possible for the simulator, since it knows α. After that, the simulator is ready to open y according to the protocol. If y passes the check, the simulator sends Deliver to $\mathcal{F}_{\text{ONLINE}}$.
- If BreakDown is true, the simulator inputs the result of the simulation to $\mathcal{F}_{\text{ONLINE}}$.

Fig. 22. Simulator for the Online phase

F Active Security

The following is a sketch of a method for an actively secure version of $\Pi_{\text{EncCommit}}$. More specifically, we assume players have access to an ideal functionality $\mathcal{F}^A_{\text{KeyGen}}$ which generates the key material as $\mathcal{F}_{\text{KeyGen}}$, but it models active security rather than covert security. More concretely, this just means that there is no "cheat option" that the adversary can choose. The purpose of this section is therefore to describe a protocol $\Pi^A_{\text{EncCommit}}$ which securely implements an ideal functionality $\mathcal{F}^A_{\text{SHE}}$ in the $\mathcal{F}^A_{\text{KeyGen}}$-hybrid model, where $\mathcal{F}^A_{\text{SHE}}$ behaves as $\mathcal{F}_{\text{SHE}}$, but, again, models active security.

The protocol is inspired by the protocol from [22] where a particularly efficient variant of the cut-and-choose approach was developed.

Let P_i be the player who is to produce ciphertexts to be verified by the other players. The protocol is parametrized by two natural numbers T, b where b divides T. We will set $t = T/b$. The protocol will produce as output t ciphertexts $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$.

Each such ciphertext is generated according to the algorithm described earlier, and is therefore created from the public key and four polynomials m, v, e_0 and e_1. To make the notation easier to deal with below, we rename these as f_1, f_2, f_3, f_4. We can then observe that there exist ρ_l, for $l = 1, \ldots, 4$ such that $\|f_l\|_\infty \leq \rho_l$ except with negligible probability. Concretely, we can use $\rho_1 = p/2$, $\rho_2 = 1$ and $\rho_3 = \rho_4 = \rho$ where ρ can be determined by a tail-bound on the gaussian distribution used for generating f_3, f_4.

The player P_i will also create a set of random *reference ciphertexts* $\mathfrak{d}_0, \ldots, \mathfrak{d}_{2T-1}$ that are used to verify that $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$ are well-formed and that P_i knows what they contain. Each $\mathfrak{d}_j$ is created from 4 polynomials $g_1, \ldots, g_4$ in the same way as above, but the polynomials are created with a different distribution. Namely, they are random subject to $\|g_i\|_\infty \leq 4 \cdot \delta \cdot \rho_i \cdot T \cdot \phi(m)$, where $\delta > 1$ is some constant.

The protocol now proceeds as follows:

1. Below P_i is given some number of attempts to prove that his ciphertexts are correctly formed. The protocol is parametrized by a number M which is the maximal number of allowed attempts. We start by setting a counter $v = 1$.
2. P_i broadcasts the ciphertexts $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$ and the reference ciphertexts $\mathfrak{d}_0, \ldots, \mathfrak{d}_{2T-1}$ containing plaintexts. These ciphertexts should be generated from seeds $s_0, \ldots, s_{2T-1}$ that are first sent through the random oracle and the output is used to generate the plaintext and randomness for the encryptions.
3. A random index subset of size T is chosen, and P_i must broadcast s_i for $i \in T$. Players check that each opened s_i indeed induces the ciphertext $\mathfrak{d}_i$, and abort if this is not the case.
4. A random permutation π on T items is generated and the unopened ciphertexts are permuted according to π. We renumber the permuted ciphertexts and call them $\mathfrak{d}_0, \ldots, \mathfrak{d}_{T-1}$.
5. Now, for each $\mathfrak{c}_i$, the subset of ciphertexts $\{\mathfrak{d}_{bi+j} |\ j = 0, \ldots, b-1\}$ is used to demonstrate that $\mathfrak{c}_i$ is correctly formed. This is called the block of ciphertexts assigned to $\mathfrak{c}_i$. We do as follows:
   (a) For each i, j do the following: let $f_1, \ldots, f_4$ and $g_1, \ldots, g_4$ be the polynomials used to form $\mathfrak{c}_i$, respectively $\mathfrak{d}_{bi+j}$. Define $z_l = f_l + g_l$, for $l = 1, \ldots, 4$.
   (b) Player P_i checks that $\|z_l\|_\infty \leq 4 \cdot \delta \cdot \rho_l \cdot T \cdot \phi(m) - \rho_l$. If this is the case, he broadcasts z_l, for $l = 1, \ldots, 4$. Otherwise he broadcasts $\perp$.
   (c) In the former case players check that $\|z_l\|_\infty$ is in range for $l = 1, \ldots, 4$ and that the z_l's induce the ciphertext $\mathfrak{c}_i + \mathfrak{d}_{bi+j}$.
   (d) At the end, players verify that for each $\mathfrak{c}_i$, P_i has correctly opened $\mathfrak{c}_i + \mathfrak{d}_{bi+j}$ for all ciphertexts in the block assigned to $\mathfrak{c}_i$.
   (e) If all checks go through, output $\mathfrak{c}_0, \ldots, \mathfrak{c}_{t-1}$ and exit. Else, if $v < M$, increment v and go to step 2. Finally, if $v = M$, the prover has failed to convince us M times, so abort the protocol.

It is possible to adapt the protocol for proving that the plaintexts in $\mathfrak{c}_i$ satisfy certain special properties. For instance, assume we want to ensure that the plaintext polynomial f_1 is a constant polynomial, i.e., only the degree-0 coefficient is non-zero. We do this by generating the reference ciphertexts such that for each $\mathfrak{d}_i$, the polynomial g_1 is also a constant polynomial. When opening we check that the plaintext polynomial is always constant. The proof of security is trivially adapted to this case.

Some intuition for why this works: after half the reference ciphertexts are opened, we know that except with exponentially small probability almost all the unopened ciphertexts are well formed. A simulator will be able to extract randomness and plaintext for all the well formed ones. When we split the unopened $\mathfrak{d}_j$'s randomly in blocks of b ciphertexts, it is therefore very unlikely that some block contains only bad ciphertexts. It can be shown that the probability that this happens is at most $t^{1-b} \cdot (e \cdot \ln(2))^{-b}$ [22].

Assume P_i is corrupt: Now, if he survives one iteration of the test, and no block was completely bad, it follows that for every $\mathfrak{c}_i$, he has opened opened at least one $\mathfrak{c}_i + \mathfrak{d}_{bi+j}$ where $\mathfrak{d}_{bi+j}$ was well formed. The simulator can therefore extract a way to open $\mathfrak{c}_i$ since $\mathfrak{c}_i = (\mathfrak{c}_i + \mathfrak{d}_{bi+j}) - \mathfrak{d}_{bi+j}$. It will be able to compute polynomials f_l for $\mathfrak{c}_i$ with $\|f_l\|_\infty \leq 8 \cdot \delta \cdot \rho_l \cdot T \cdot \phi(m)$. Therefore, if some $\mathfrak{c}_i$ is not of this form, the prover can survive one iteration of the test with probability at most $t^{1-b} \cdot (e \cdot \ln(2))^{-b}$. To survive the entire protocol, the prover needs to win in at least one of the M iterations, and this happens with probability at most $M \cdot t^{1-b} \cdot (e \cdot \ln(2))^{-b}$, by the union bound.

Assume P_i is honest: Then when he decides whether to open a given ciphertext, the probability that a single coefficient is in range is $\frac{1}{4 \cdot \delta \cdot \phi(m) \cdot T}$. There are $4 \cdot \phi(m)$ coefficients in a single ciphertext and up to T ciphertexts to open, so by a union bound, P_i will not need to send $\perp$ at all, except with probability $1/\delta$. The probability that an honest prover fails to complete the protocol is hence $(1/\delta)^M$. We therefore see that the completeness error vanishes exponentially with increasing M, and in the soundness probability, we only loose $\log M$ bits of security.

It is easy to see that for each opening done by an honest prover, the polynomials z_l will have coefficients that are uniformly distributed in the expected range, so the protocol can be simulated.

Finally, note that in a normal run of the protocol, only 1 iteration is required, except with probability $1/\delta$. So in practice, what counts for the efficiency is the time we spend on one iteration.

In our experiments we implemented the above protocol with the following parameter choices $\delta = 256$, $M = 5$, $t = 12$ and $b = 16$. This guaranteed a cheating probability of 2^{-40}, as well as the probability of an honest prover failing of 2^{-40}. In addition the choice of $t = 12$ was to ensure that each run of the protocol created enough ciphertexts to be run in two executions of the main loop of the multiplication triple production protocol. By increasing t and decreasing b one can improve the amortized complexity of the protocol while keeping the error probabilities the same. This comes at the cost of increased memory usage, primarily because decreasing b to, e.g, $b/2$ means that t needs to be replaced by essentially t^2. On our test machines $t = 12$ seemed to provide the best compromise.

G Parameters of the BGV Scheme

In this appendix we present an analysis of the parameters needed by the BGV to ensure that the distributed decryption procedure can decrypt the ciphertexts produced in the offline phase and that the scheme is "secure". Unlike in [14], which presents the analysis in terms of a worst case analysis, we use the expected case analysis used in [16].

G.1 Expected Values of Norms

Given an element $a \in R$ (represented as a polynomial) we define $\|a\|_p$ to be the standard p-norm of the coefficient vector (usually for $p = 1, 2$ or ∞). We also define $\|a\|_p^{\mathsf{can}}$ to be the p-norm of the same element when mapped into the canonical embedding i.e.

$$\|a\|_p^{\mathsf{can}} = \|\kappa(a)\|_p$$

where $\kappa(a) : R \longrightarrow \mathbb{C}^{\phi(m)}$ is the canonical embedding. The key two relationships are that

$$\|a\|_\infty \leq c_m \cdot \|a\|_\infty^{\mathsf{can}} \text{ and } \|a\|_\infty^{\mathsf{can}} \leq \|a\|_1,$$

for some constant c_m depending on m. Since in our protocol we select m to be a power of two then we have $c_m = 1$.

We also define the *canonical embedding norm reduced modulo* q of an element $a \in R$ as the smallest canonical embedding norm of any a' which is congruent to a modulo q. We denote it as

$$|a|_q^{can} = \min\{\ \|a'\|_\infty^{\mathsf{can}} :\ a' \in R,\ a' \equiv a \pmod{q}\ \}.$$

We sometimes also denote the polynomial where the minimum is obtained by $[a]_q^{\mathsf{can}}$, and call it the *canonical reduction* of a modulo q.

Following [16][Appendix A.5] we examine the variances of the different distributions utilized in our protocol. Let ζ_m denote any complex primitive m-th root of unity. Sampling $a \in R$ from $\mathcal{HWT}(h, \phi(m))$ and looking at $a(\zeta_m)$ produces a random variable with variance h, when sampled from $\mathcal{ZO}(0.5, \phi(m))$ we obtain variance $\phi(m)/2$, when sampled from $\mathcal{DG}(\sigma^2, \phi(m))$ we obtain variance $\sigma^2 \cdot \phi(m)$ and when sampled from $\mathcal{U}(q, \phi(m))$ we obtain variance $q^2 \cdot \phi(m)/12$. By the law of large numbers we can use $6 \cdot \sqrt{V}$, where V is the above variance, as a high probability bound on the size of $a(\zeta_m)$, and this provides a bound on the canonical embedding norm of a.

If we take a product of two, three, or four such elements with variances $V_1, V_2, \ldots, V_4$ we use $16 \cdot \sqrt{V_1 \cdot V_2}$, $9.6 \cdot \sqrt{V_1 \cdot V_2 \cdot V_3}$ and $7.3 \cdot \sqrt{V_1 \cdot V_2 \cdot V_3 \cdot V_4}$ as the resulting bounds since

$$\mathsf{erfc}(4)^2 \approx \mathsf{erfc}(3.1)^3 \approx \mathsf{erfc}(2.7)^4 \approx 2^{-50}.$$

G.2 Key Generation

We first need to establish the rough distributions (i.e. variances) of the resulting keys arising from our key generation procedure. For our purposes we are only interested in the variance of the associated distributions in the canonical embedding, in which case we obtain

$$\begin{aligned}
\mathsf{Var}(\kappa(\mathfrak{s}_j)) &= n \cdot \mathsf{Var}(\kappa(\mathfrak{s}_{i,j})) = n \cdot h, \\
\mathsf{Var}(\kappa(a_j)) &= q_1^2 \cdot \phi(m)/12, \\
\mathsf{Var}(\kappa(\epsilon_j)) &= n \cdot \mathsf{Var}(\kappa(\epsilon_{i,j})) = n \cdot \sigma^2 \cdot \phi(m).
\end{aligned}$$

We will also need to analyze the distributions of the randomness needed to produce $\mathfrak{enc}_j$. Here we assume that all parties follow the protocol and we are only interested in the output final extended public key, thus we write (dropping the j to avoid overloading the reader)

$$\mathfrak{enc} = (b_{\mathfrak{s},\mathfrak{s}^2}, a_{\mathfrak{s},\mathfrak{s}^2})$$

where

$$b_{\mathfrak{s},\mathfrak{s}^2} = a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} + p \cdot e_{\mathfrak{s},\mathfrak{s}^2} - p_1 \cdot \mathfrak{s}^2.$$

We can also write

$$\begin{aligned}
\mathfrak{enc}_i' &= (b \cdot v_i + p \cdot e_{0,i} - p_1 \cdot \mathfrak{s}_i, a \cdot v_i + p \cdot e_{1,i}) \\
\mathfrak{zero}_i &= (b \cdot v_i' + p \cdot e_{0,i}', a \cdot v_i' + p \cdot e_{1,i}')
\end{aligned}$$

where $(v_i, e_{0,i}, e_{1,i}) \leftarrow \mathcal{RC}_s(0.5, \sigma^2, \phi(m))$ and $(v_i', e_{0,i}', e_{1,i}') \leftarrow \mathcal{RC}_s(0.5, \sigma^2, \phi(m))$. We therefore have

$$a_{\mathfrak{s},\mathfrak{s}^2} = \sum_{i=1}^{n} \mathfrak{s}_i \cdot \left(\sum_{j=1}^{n} a \cdot v_j + p \cdot e_{1,j} \right) + \sum_{i=1}^{n} (a \cdot v_i' + p \cdot e_{1,i}'),$$

and

$$\begin{aligned}
b_{\mathfrak{s},\mathfrak{s}^2} &= \sum_{i=1}^{n} \mathfrak{s}_i \cdot \left(\sum_{j=1}^{n} b \cdot v_j + p \cdot e_{0,j} - p_1 \cdot \mathfrak{s}_j \right) + \sum_{i=1}^{n} (b \cdot v'_i + p \cdot e'_{0,i}) \\
&= a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} - \mathfrak{s} \cdot \sum_{i=1}^{n} \mathfrak{s}_i \cdot \left(\sum_{j=1}^{n} a \cdot v_j + p \cdot e_{1,j} \right) + \sum_{i=1}^{n} \mathfrak{s}_i \cdot \left(\sum_{j=1}^{n} b \cdot v_j + p \cdot e_{0,j} - p_1 \cdot \mathfrak{s}_j \right) \\
&\quad + \sum_{i=1}^{n} \left((a \cdot \mathfrak{s} + p \cdot \epsilon) \cdot v'_i + p \cdot e'_{0,i} \right) - \mathfrak{s} \cdot \sum_{i=1}^{n} (a \cdot v'_i + p \cdot e'_{1,i}) \\
&= a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} + \sum_{i=1}^{n} \left(\sum_{j=1}^{n} b \cdot v_j \cdot \mathfrak{s}_i + p \cdot e_{0,j} \cdot \mathfrak{s}_i - p_1 \cdot \mathfrak{s}_i \cdot \mathfrak{s}_j - \mathfrak{s} \cdot \mathfrak{s}_i \cdot a \cdot v_j - \mathfrak{s} \cdot \mathfrak{s}_i \cdot p \cdot e_{1,j} \right) \\
&\quad + p \cdot \sum_{i=1}^{n} \left(\epsilon \cdot v'_i + e'_{0,i} - e'_{1,i} \cdot \mathfrak{s} \right) \\
&= a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} + p \cdot \sum_{i=1}^{n} \left(\sum_{j=1}^{n} (\epsilon \cdot v_j \cdot \mathfrak{s}_i + e_{0,j} \cdot \mathfrak{s}_i - \mathfrak{s} \cdot \mathfrak{s}_i \cdot e_{1,j}) + \epsilon \cdot v'_i + e'_{0,i} - e'_{1,i} \cdot \mathfrak{s} \right) - p_1 \cdot \mathfrak{s}^2 \\
&= a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} + p \cdot e_{\mathfrak{s},\mathfrak{s}^2} - p_1 \cdot \mathfrak{s}^2
\end{aligned}$$

where

$$e_{\mathfrak{s},\mathfrak{s}^2} = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} (\epsilon \cdot v_j \cdot \mathfrak{s}_i + e_{0,j} \cdot \mathfrak{s}_i - \mathfrak{s} \cdot \mathfrak{s}_i \cdot e_{1,j}) + \epsilon \cdot v'_i + e'_{0,i} - e'_{1,i} \cdot \mathfrak{s} \right). \tag{2}$$

Thus the values $\mathfrak{enc}$ are indeed genuine "quasi-encryptions" of $-p_1 \cdot \mathfrak{s}^2$ with respect to the secret key $\mathfrak{s}$ and the modulus q_1. Equation 2 will be used later to establish the properties of the output of the SwitchKey procedure.

G.3 BGV Procedures

We can now turn to each of the procedures in turn of the two level BGV scheme we are using and estimate the output noise term. For a ciphertext $\mathfrak{c} = (c_0, c_1, \ell)$ we define the "noise" to be an upper bound on the value

$$\|c_0 - \mathfrak{s} \cdot c_1\|_\infty^{\mathsf{can}}.$$

$\underline{\mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m})}$: Given a fresh ciphertext $(c_0, c_1, 1)$, we calculate a bound (with high probability) on the output noise by

$$\begin{aligned}
\|c_0 - \mathfrak{s} \cdot c_1\|_\infty &\leq \|c_0 - \mathfrak{s} \cdot c_1\|_\infty^{\mathsf{can}} \\
&= \|((a \cdot \mathfrak{s} + p \cdot \epsilon) \cdot v + p \cdot e_0 + \mathbf{m} - (a \cdot v + p \cdot e_1) \cdot \mathfrak{s}\|_\infty^{\mathsf{can}} \\
&= \|\mathbf{m} + p \cdot (\epsilon \cdot v + e_0 - e_1 \cdot \mathfrak{s})\|_\infty^{\mathsf{can}} \\
&\leq \|\mathbf{m}\|_\infty^{\mathsf{can}} + p \cdot \left(\|\epsilon \cdot v\|_\infty^{\mathsf{can}} + \|e_0\|_\infty^{\mathsf{can}} + \|e_1 \cdot \mathfrak{s}\|_\infty^{\mathsf{can}} \right) \\
&\leq \phi(m) \cdot p/2 + p \cdot \sigma \cdot \left(16 \cdot \phi(m) \cdot \sqrt{n/2} + 6 \cdot \sqrt{\phi(m)} + 16 \cdot \sqrt{n \cdot h \cdot \phi(m)} \right) = B_{\mathsf{clean}}.
\end{aligned}$$

Note this value of B_{clean} is different from that in [16] due to the different distributions resulting from the distributed key generation.

$\underline{\mathsf{SwitchModulus}((c_0, c_1), \ell)}$: If the input ciphertext has noise ν then the output ciphertext will have noise ν' where

$$\nu' = \frac{\nu}{p_\ell} + B_{\mathsf{scale}}.$$

The value B_{scale} is an upper bound on the quantity $\|\tau_0 + \tau_1 \cdot \mathfrak{s}\|_\infty^{\mathsf{can}}$, where $\kappa(\tau_i)$ is drawn from a distribution which is close to a complex Gaussian with variance $\phi(m) \cdot p^2/12$. We therefore, we can (with high probability) take the upper bound to be

$$\begin{aligned} B_{\mathsf{scale}} &= 6 \cdot p \cdot \sqrt{\phi(m)/12} + 16 \cdot p \cdot \sqrt{n \cdot \phi(m) \cdot h/12}, \\ &= p \cdot \sqrt{3 \cdot \phi(m)} \cdot \left(1 + 8 \cdot \sqrt{n \cdot h}/3\right). \end{aligned}$$

Again, note the dependence on n (compared to [16]) as the secret key $\mathfrak{s}$ is selected from a distribution with variance $n \cdot h$, and not just h. Also note the dependence on p due to the plaintext space being defined mod p as opposed to mod 2 in [16].

$\underline{\mathsf{Dec}_{\mathfrak{s}}(\mathfrak{c})}$: As explained in [14, 16] this procedure works when the noise ν associated with a ciphertext satisfies $\nu = c_m \cdot \nu < q_\ell/2$.

$\underline{\mathsf{DistDec}_{\mathfrak{s}_i}(\mathfrak{c})}$: The value B is an upper bound on the noise ν associated with a ciphertext we will decrypt in our protocols. To ensure valid distributed decryption we require

$$2 \cdot (1 + 2^{\mathsf{sec}}) \cdot B < q_\ell.$$

Given a value of B, we therefore will obtain a lower bound on p_0 by the above inequality. The addition of a random term with infinity norm bounded by $2^{\mathsf{sec}} \cdot B/(n \cdot p)$ in the distributed decryption procedure ensures that the individual *coefficients* of the sum $\mathbf{t}_1 + \cdots + \mathbf{t}_n$ are statistically indistinguishable from random, with probability $2^{-\mathsf{sec}}$. This does not imply that the adversary has this probability of distinguishing the simulated execution in [14] from the real execution; since each run consists of the exchange of $\phi(m)$ coefficients, and the protocol is executed many times over the execution of the whole protocol. We however feel that setting concentrating solely on the statistical indistinguishability of the coefficients is valid in a practical context.

$\underline{\mathsf{SwitchKey}(d_0, d_1, d_2)}$: In order to estimate the size of the output noise term we need first to estimate the size of the term

$$\|p \cdot d_2 \cdot \epsilon_{\mathfrak{s},\mathfrak{s}^2}\|_\infty^{\mathsf{can}}.$$

Using Equation 2 we find

$$\begin{aligned} \|p \cdot d_2 \cdot e_{\mathfrak{s},\mathfrak{s}^2})\|_\infty^{\mathsf{can}}/q_0 \leq p \cdot \sqrt{\frac{\phi(m)}{12}} \cdot \Big[& n^2 \cdot \sigma \cdot \Big(7.3 \cdot \sqrt{n \cdot h \cdot \phi(m)^2/2} + 9.6 \cdot \sqrt{h \cdot \phi(m)} \\ & \qquad + 7.3 \cdot h \cdot \sqrt{n \cdot \phi(m)}\Big) \\ & + n \cdot \Big(9.6 \cdot \sigma \cdot \sqrt{n \cdot \phi(m)^2/2} + 16 \cdot \sigma \cdot \sqrt{\phi(m)} \\ & \qquad + 7.6 \cdot \sigma \cdot \sqrt{\phi(m) \cdot n \cdot h}\Big)\Big] \\ \leq p \cdot \phi(m) \cdot \sigma \cdot \Big[& n^{2.5} \cdot (1.49 \cdot \sqrt{h \cdot \phi(m)} + 2.11 \cdot h) + 2.77 \cdot n^2 \cdot \sqrt{h} \\ & + n^{1.5} \cdot (1.96 \cdot \sqrt{\phi(m)} + 2.77 \cdot \sqrt{h}) + 4.62 \cdot n\Big] \\ = B_{\mathsf{KS}}. \end{aligned}$$

Then if the input to SwitchKey has noise bounded by ν then the output noise value will be bounded by

$$\nu + \frac{B_{\mathsf{KS}} \cdot q_0}{p_1} + B_{\mathsf{scale}}.$$

$\underline{\mathsf{Mult}(\mathfrak{c}, \mathfrak{c}')}$: Combining the all the above, if we take two ciphertexts of level one with input noise bounded by ν and ν', the output noise level from multiplication will be bounded by

$$\nu'' = \left(\frac{\nu}{p_1} + B_{\mathsf{scale}}\right) \cdot \left(\frac{\nu'}{p_1} + B_{\mathsf{scale}}\right) + \frac{B_{\mathsf{KS}}}{p_1} + B_{\mathsf{scale}}.$$

G.4 Application to the Offline Phase

In all of our protocols we will only be evaluating the following circuit: We first add n ciphertexts together and perform a multiplication, giving a ciphertext with respect to modulus p_0 with noise

$$U_1 = \left(\frac{n \cdot B_{\mathsf{clean}}}{p_1} + B_{\mathsf{scale}} \right)^2 + \frac{B_{\mathsf{KS}} \cdot p_0}{p_2} + B_{\mathsf{scale}}.$$

We then add on another n ciphertexts, which are added at level one and then reduced to level zero. We therefore obtain a final upper bound on the noise for our adversarially generated ciphertexts of

$$U_2 = U_1 + \frac{n \cdot B_{\mathsf{clean}}}{p_1} + B_{\mathsf{scale}}.$$

To ensure valid (distributed) decryption, we require

$$2 \cdot U_2 \cdot (1 + 2^{\mathsf{sec}}) < p_0,$$

i.e. we take $B = U_2$ in our distributed decryption protocol.

This ensure valid decryption in our offline phase, however we still need to select the parameters to ensure security. Following the analysis in [16] of the BGV scheme we set, for 128-bit security,

$$\phi(m) \geq 33.1 \cdot \log \left(\frac{q_1}{\sigma} \right).$$

Combining the various inequalities together; a search of the parameter space the fixed values of $\sigma = 3.2$, $\mathsf{sec} = 40$ and $h = 64$, for several choices of p, n yields the estimates in tables 4, 5 and 6. And it is these parameter sizes which we use to generate the primes and rings in our implementation.

n	$\phi(m)$	$\log_2 p_0$	$\log_2 p_1$	$\log_2 q_1$	$\log_2(U_2)$
2	8192	130	104	234	89
3	8192	132	104	236	90
4	8192	132	104	236	91
5	8192	132	106	238	90
6	8192	132	106	238	91
7	8192	132	108	240	91
8	8192	132	108	240	91
9	8192	132	110	242	91
10	8192	132	110	242	91
20	8192	134	110	244	93
50	8192	136	114	250	94
100	8192	136	116	252	95

Table 4. Parameters for $p \approx 2^{32}$.

n	$\phi(m)$	$\log_2 p_0$	$\log_2 p_1$	$\log_2 q_1$	$\log_2(U_2)$
2	16384	196	136	332	154
3	16384	196	138	334	154
4	16384	196	140	336	155
5	16384	196	142	338	155
6	16384	198	140	338	156
7	16384	198	140	338	156
8	16384	198	140	338	157
9	16384	198	142	340	156
10	16384	198	142	340	156
20	16384	198	146	344	157
50	16384	200	148	348	158
100	16384	202	150	352	160

Table 5. Parameters for $p \approx 2^{64}$.

n	$\phi(m)$	$\log_2 p_0$	$\log_2 p_1$	$\log_2 q_1$	$\log_2(U_2)$
2	32768	324	202	526	283
3	32768	326	202	528	285
4	32768	326	204	530	284
5	32768	326	204	530	285
6	32768	326	206	532	284
7	32768	326	206	532	285
8	32768	326	208	534	285
9	32768	326	208	534	285
10	32768	326	208	534	285
20	32768	328	210	538	286
50	32768	330	212	542	289
100	32768	330	216	546	288

Table 6. Parameters for $p \approx 2^{128}$.

Dishonest Majority Multi-Party Computation for Binary Circuits

Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart

Dept. Computer Science, University of Bristol, United Kingdom
Enrique.LarraiadeVega@bristol.ac.uk,Emmanuela.Orsini@bristol.ac.uk,nigel@cs.bris.ac.uk

Abstract. We extend the Tiny-OT two party protocol of Nielsen et al (CRYPTO 2012) to the case of n parties in the dishonest majority setting. This is done by presenting a novel way of transferring pairwise authentications into global authentications. As a by product we obtain a more efficient manner of producing globally authenticated shares, which in turn leads to a more efficient two party protocol than that of Nielsen et al.

Keywords: Secure Multi-party Computation, Message Authentication Code, Oblivious Transfer

1 Introduction

In recent years actively secure MPC has moved from a theoretical subject into one which is becoming more practical. In the variants of multi-party computation which are based on secret sharing the major performance improvement has come from the technique of authenticating the shared data and/or the shares themselves using information theoretic message authentication codes (MACs). This idea has been used in a number of works: In the case of two-party MPC for binary circuits in [13], for n-party dishonest majority MPC for arithmetic circuits over a "largish" finite field [4,7], and for n-party dishonest majority MPC over binary circuits [8]. All of these protocols are in the pre-processing model, in which the parties first engage in a function and input independent offline phase. The offline phase produces various pieces of data, often Beaver style [3] "multiplication triples", which are then consumed in the online phase when the function is determined and evaluated.

In the case of the protocol of [13], called Tiny-OT in what follows, the authors use the technique of applying information theoretic MACs to the oblivious transfer (OT) based GMW protocol [10] in the two party setting. In this protocol the offline phase consists of producing a set of pre-processed random OTs which have been authenticated. The offline phase is then executed efficiently using a variant of the OT extension protocol of [12]. For a detailed discussion on OT extension see [2,12,13]. In this work we shall take OT extension as a given sub-procedure.

For the case of n party protocols, where $n > 2$, there are three main techniques using such MACs. In [4] each share of a given secret is authenticated by pairwise MACs, i.e. if party P_i holds a share a_i, then it will also hold a MAC $M_{i,j}$ for every $j \neq i$, and party P_j will hold a key $K_{i,j}$. Then, when the value a_i is made public, party P_i also reveals the $n-1$ MAC values, that are then checked by other parties using their private keys $K_{i,j}$. Note that each pair of parties holds a separate key/MAC for each share value. In [7] the authors obtain a more efficient online protocol by replacing the MACs from [4] with global MACs which authenticate the shared values a, as opposed to the shares themselves. The authentication is also done with respect to a fixed global MAC key (and not pairwise and data dependent). This method was improved in [6], where it is shown how to verify these global MACs without revealing the secret global key. In [8] the authors adapt the technique from [7] for the case of small finite fields, in a way which allows one to authenticate multiple field elements at the same time, without requiring multiple MACs. This is performed

using a novel application of ideas from coding theory, and results in a reduced overhead for the online phase.

One can think of the Tiny-OT protocol as applying the authentication technique of [4] to the two party, binary circuit case, with a pre-processing which is based on OT as opposed to semi-homomorphic encryption. For two party protocols over binary circuits practical experiments show that Tiny-OT far out-performs other protocols, such as those based on Yao's garbled circuit technique. This is because of the performance of the offline phase of the Tiny-OT protocol. Thus a natural question is to ask, whether one can extend the Tiny-OT protocol to the n-party setting for binary circuits.

Results and Techniques In this paper we mainly address ourselves to the above question, i.e. how can we generalize the two-party protocol from [13] to the n-party setting?

We first describe what are the key technical difficulties we need to overcome. The Tiny-OT protocol at its heart has a method for authenticating random bits via pairwise MACs, which itself is based on an efficient protocol for OT-extension. In [13] this protocol is called aBit. Our aim is to use this efficient two-party process as a black-box. Unfortunately, if we extend this procedure naively to the three party case, we would obtain (for example) that parties P_1 and P_2 could execute the protocol so that P_1 obtains a random bit and a MAC, whilst P_2 obtains a key for the MAC used to authenticate the random bit. However, party P_3 obtains no authentication on the random bit obtained by P_1, nor does it obtain any information as to the MAC or the key.

To overcome this difficulty, we present a protocol in which we fix an unknown global random key and where each party holds a share of this key. Then by executing the pairwise aBit protocol, we are able to obtain a secret shared value, as well as a shared MAC, by all n-parties. This resulting MAC is identical to the MAC used in the SPDZ protocol from [6]. This allows us to obtain authenticated random shares, and in addition to permit parties to enter their inputs into the MPC protocol.

The online phase will then follow similarly to [6], if we can realize a protocol to produce "multiplication triples". In [13] one can obtain such triples by utilizing a complex method to produce authenticated random OTs and authenticated random ANDs (called aOTs and aANDs)[1]. We notice that our method for obtaining authenticated bits also enables us to obtain a form of authenticated OTs in a relatively trivial manner, and such authenticated OTs can be used directly to implement a multiplication gate in the online phase.

Our contribution is twofold. First, we generalize the two-party Tiny-OT protocol to the n-party setting, using a new technique for authentication of secret shared bits, and new offline and online phases. Thus we are able to dispense with the protocols to generate aOTs and aANDs from [13]. Second, and as a by product, we obtain a more efficient protocol than the original Tiny-OT protocol, in the two party setting when one measures efficiency in terms of the number of aBit's needed per multiplication gate.

The security of our protocols are proven in the standard universal composability (UC) framework [5] against a malicious adversary and static corruption of parties.

We end this introduction by describing two possible extensions to our work. Firstly, each bit in our protocol is authenticated by an element in a finite field $\mathbb{F}_{2^\kappa}$. Whilst such values are never transmitted in our online phase due to our $\mathsf{MACCheck}$ protocol, they do provide an overhead in the

[1] In fact the paper [13] does not produce such multiplication triples, but they follow immediately from the presentation in the paper and would result in a more efficient online phase than that described in [13]

computation. In [8] the authors show how to reduce this overhead using coding theory techniques. It would be interesting to see how such techniques could be applied to our protocol, and what advantage if any they would bring.

Secondly, our protocol requires $n \cdot (n-1)/2$ executions of the aBit protocol from [13]. Each pairwise invocation requires the execution of an OT-extension protocol, and hence we require $O(n^2)$ such OT-channels. In [11], in the context of traditional MPC protocols, the authors present techniques and situations in which the number of OT-channels can be reduced to $O(n)$. It would be interesting to see how such techniques could be applied in practice to the protocol described in this paper.

2 Notation

In this section we settle the notation used throughout the paper. We use κ to denote the security parameter. We let $\mathsf{negl}(\kappa)$ denote some unspecified function $f(\kappa)$, such that $f = o(\kappa^{-c})$ for every fixed constant c, saying that such a function is *negligible* in κ. We say that a probability is *overwhelming* in κ if it is $1 - \mathsf{negl}(\kappa)$.

We consider the sets $\{0,1\}$ and $\mathbb{F}_2^\kappa$ endowed with the structure of the fields $\mathbb{F}_2$ and $\mathbb{F}_{2^\kappa}$, respectively. Let $\mathbb{F} = \mathbb{F}_{2^\kappa}$, we will denote elements in $\mathbb{F}$ with greek letters and elements in $\mathbb{F}_2$ with roman letters.

We will additively secret share bits and elements in $\mathbb{F}$, among a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, and sometimes abuse notation identifying subsets $\mathcal{I} \subseteq \{1, \ldots, n\}$ with the subset of parties indexed by $i \in \mathcal{I}$. We write $\langle a \rangle^{\mathcal{I}}$ if a is shared amongst the set $\mathcal{I} = \{i_1, \ldots, i_t\}$ with party P_{i_j} holding a value a_{i_j}, such that $\sum_{i_j \in \mathcal{I}} a_{i_j} = a$. Also, if an element $x \in \mathbb{F}_2$ (resp. $\beta \in \mathbb{F}$) is additively shared among *all* parties we write $\langle x \rangle$ (resp. $\langle \beta \rangle$). We adopt the convention that if $a \in \mathbb{F}_2$ (resp. $\beta \in \mathbb{F}$) then the shares $a_i \in \mathbb{F}_2$ (resp. $\beta_i \in \mathbb{F}$).

(Linear) arithmetic on the $\langle \cdot \rangle^{\mathcal{I}}$ sharings can be performed as follows. Given two sharings $\langle x \rangle^{\mathcal{I}_x} = \{x_{i_j}\}_{i_j \in \mathcal{I}_x}$ and $\langle y \rangle^{\mathcal{I}_y} = \{y_{i_j}\}_{i_j \in \mathcal{I}_y}$ we can compute the following linear operations

$$\begin{aligned} a \cdot \langle x \rangle^{\mathcal{I}_x} &= \{a \cdot x_{i_j}\}_{i_j \in \mathcal{I}_x}, \\ a + \langle x \rangle^{\mathcal{I}_x} &= \{a + x_{i_1}\} \cup \{x_{i_j}\}_{i_j \in \mathcal{I}_x \setminus \{i_1\}}, \\ \langle x \rangle^{\mathcal{I}_x} + \langle y \rangle^{\mathcal{I}_y} &= \langle x + y \rangle^{\mathcal{I}_x \cup \mathcal{I}_y} \\ &= \{x_{i_j}\}_{i_j \in \mathcal{I}_x \setminus \mathcal{I}_y} \cup \{y_{i_j}\}_{i_j \in \mathcal{I}_y \setminus \mathcal{I}_x} \cup \{x_{i_j} + y_{i_j}\}_{i_j \in \mathcal{I}_x \cap \mathcal{I}_y}. \end{aligned}$$

Our protocols will make use of pseudo-random functions, which we will denote by $\mathsf{PRF}_s^{X,t}(\cdot)$ where for a key s and input $m \in \{0,1\}^*$ the pseudo-random function is defined by $\mathsf{PRF}_s^{X,t}(m) \in X^t$, where X is some set and t is a non-negative integer.

Authentication of Secret Shared Values. As described in the introduction the literature gives two ways to authenticate a secret globally held by a system of parties, one is to authenticate the shares of each party, as in [4], the other is to authenticate the secret itself, as in [7]. In addition we can also have authentication in a pairwise manner, as in [4,13], or in a global manner, as in [7]. Both combinations of these variants can be applied, but each implies important practical differences, e.g., the total amount of data each party needs to store and how checking of the MACs is performed. In this work we will use a combination of different techniques, indeed the main technical trick is a method to pass from the technique used in [13] to the technique used in [7].

Our main technique for authentication of secret shared bits is applied by placing an *information theoretic tag* (MAC) on the shared bit x. The authenticating key is a random line in $\mathbb{F}$, and the MAC on x is its corresponding line point, thus, the linear equation $\mu_\delta(x) = \nu_\delta(x) + x \cdot \delta$ holds, for some $\mu_\delta(x), \nu_\delta(x), \delta \in \mathbb{F}$. We will use these lines in various operations[2], for various values of δ. In particular, there will be a special value of δ, which we denote by α and assume to be $\langle \alpha \rangle^{\mathcal{P}}$ shared, which represents the *global* key for our online MPC protocol. This will be the same key for every bit that needs to be authenticated. It will turn out that for the key α we always have $\nu_\alpha(x) = 0$. By abuse of notation we will sometimes refer to a general δ also as a *global* key, and then the corresponding $\nu_\delta(x)$, is called the *local* key.

Distinguishing between parties, say $\mathcal{I}$, that can reconstruct bits (together with the line point), and those parties, say $\mathcal{J}$, that can reconstruct the line gives a natural generalization of both ways to authenticate, and it also allows to move easily from one to another. We write $[x]^{\mathcal{I}}_{\delta,\mathcal{J}}$ if there exist $\mu_\delta(x), \nu_\delta(x) \in \mathbb{F}$ such that:

$$\mu_\delta(x) = \nu_\delta(x) + x \cdot \delta,$$

where we have that x and $\mu_\delta(x)$ are $\langle \cdot \rangle^{\mathcal{I}}$ shared, and $\nu_\delta(x)$ and δ are $\langle \cdot \rangle^{\mathcal{J}}$ shared, i.e. there are values x_i, μ_i, and ν_j, δ_j, such that

$$x = \sum_{i \in \mathcal{I}} x_i, \qquad \mu_\delta(x) = \sum_{i \in \mathcal{I}} \mu_i, \qquad \nu_\delta(x) = \sum_{j \in \mathcal{J}} \nu_j, \qquad \delta = \sum_{j \in \mathcal{J}} \delta_j.$$

Notice that $\mu_\delta(x)$ and $\nu_\delta(x)$ depend on δ and x: we can fix δ and so obtain *key-consistent* representations of bits, or we can fix x and obtain different *key-dependant* representations for the same bit x. To ease the reading, we drop the sub-index $\mathcal{J}$ if $\mathcal{J} = \mathcal{P}$, and, also, the dependence on δ and x when it is clear from the context. We note that in the case of $\mathcal{I}_x = \mathcal{J}_x$ then we can assume $\nu_j = 0$.

When we take the fixed global key α and we have $\mathcal{I}_x = \mathcal{J}_x = \mathcal{P}$, we simplify notation and write $[\![x]\!] = [x]^{\mathcal{P}}_{\alpha,\mathcal{P}}$. By our comment above we can, in this situation, set $\nu_j = 0$ [3], this means that a $[\![x]\!]$ sharing is given by two sharings $\big(\langle x \rangle^{\mathcal{P}}, \langle \mu \rangle^{\mathcal{P}}\big)$. Notice that the $[\![\cdot]\!]$-representation of a bit x implies that x is *authenticated* with the global key α and that it is $\langle \cdot \rangle$-shared, i.e. its value is actually unknown to the parties.

This notation does not quite align with the previous secret sharing schemes used in the literature, but it is useful for our purposes. For example, with this notation the MAC scheme of [4] is one where each data element x is shared via $[x_i]^i_{\alpha_j,j}$ sharings. Thus the data is shared via a $\langle x \rangle$ sharing and the authentication is performed via $[x_i]^i_{\alpha_j,j}$ sharings, i.e. we are using two sharing schemes simultaneously. In [7] the data is shared via our $[\![x]\!]$ notation, except that the MAC key value ν is set equal to $\nu = \nu'/\alpha$, where ν' being a *public value*, as opposed to a shared value. Our $[\![x]\!]$ sharing is however identical to that used in [6], bar the differences in the underlying finite fields.

Looking ahead we say that a bit $[\![x]\!]$ is *partially opened* if $\langle x \rangle$ is opened, i.e. the parties reveal the shares of x, but not the shares of the MAC value $\mu_\alpha(x)$.

Arithmetic on $[\![x]\!]$ Shared Values. Given two representations $[x]^{\mathcal{I}_x}_{\delta,\mathcal{J}_x} = \big(\langle x \rangle^{\mathcal{I}_x}, \langle \mu_\delta(x) \rangle^{\mathcal{I}_x}, \langle \nu_\delta(x) \rangle^{\mathcal{J}_x}\big)$ and $[y]^{\mathcal{I}_y}_{\delta,\mathcal{J}_y} = \big(\langle y \rangle^{\mathcal{I}_y}, \langle \mu_\delta(y) \rangle^{\mathcal{I}_y}, \langle \nu_\delta(y) \rangle^{\mathcal{J}_y}\big)$, under same the δ, the parties can locally compute $[x +$

[2] For example, we will also use lines to generate OT-tuples, i.e. quadruples of authenticated bits which satisfy the algebraic equation for a random OT.

[3] Otherwise one can subtract ν_j from μ_j, before setting ν_j to zero.

$y]^{\mathcal{I}_x \cup \mathcal{I}_y}_{\delta, \mathcal{J}_x \cup \mathcal{J}_y}$ as $\big(\langle x\rangle^{\mathcal{I}_x} + \langle y\rangle^{\mathcal{I}_y}, \langle \mu_\delta(x)\rangle^{\mathcal{I}_x} + \langle \mu_\delta(y)\rangle^{\mathcal{I}_y}, \langle \nu_\delta(x)\rangle^{\mathcal{J}_x} + \langle \nu_\delta(y)\rangle^{\mathcal{J}_y}\big)$ using the arithmetic on $\langle\cdot\rangle^{\mathcal{I}}$ sharings above.

Let $[\![x]\!] = \big(\langle x\rangle, \langle \mu(x)\rangle\big)$ and $[\![y]\!] = \big(\langle y\rangle, \langle \mu(y)\rangle\big)$ be two different authenticated bits. Since our sharings are linear, as well as the MACs, it is easy to see that the parties can locally perform linear operations:

$$\begin{aligned}
[\![x]\!] + [\![y]\!] &= \big(\langle x\rangle + \langle y\rangle,\ \langle \mu(x)\rangle + \langle \mu(y)\rangle\big) = [\![x+y]\!] \\
a \cdot [\![x]\!] &= \big(a \cdot \langle x\rangle,\ a \cdot \langle \mu(x)\rangle\big) = [\![a \cdot x]\!], \\
a + [\![x]\!] &= \big(a + \langle x\rangle, \langle \mu(a+x)\rangle\big) = [\![a+x]\!].
\end{aligned}$$

where $\langle \mu(a+x)\rangle$ is the sharing obtained by each party $i \in \mathcal{P}$ holding the value $\alpha_i \cdot a + \mu_i(x)$.

This means that the only remaining question to enable MPC on $[\![\cdot]\!]$-shared values is how to perform multiplication and how to generate the $[\![\cdot]\!]$-shared values in the first place. Note, that a party P_i that wishes to enter a value into the MPC computation is wanting to obtain a $[x]^i_{\alpha,\mathcal{P}}$ sharing of its input value x, and that this is a $[\![x]\!]$-representation if we set $x_i = x$ and $x_j = 0$ for $j \neq i$.

3 MPC Protocol for Binary Circuit

We start presenting a high level view of the protocols that allow us to perform multi-party computation for binary circuits. We assume synchronous communication and authentic point-to-point channels. Our protocol is in the pre-processing model in which we allow a function (and input) independent pre-processing, or offline, phase which produces correlated randomness. This enables a lightweight online phase, that does not need public-key machinery.

In the following sections we will describe a protocol, Π_{Online}, implementing the actual function evaluation in the $(\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Prep}})$-hybrid model; a protocol, Π_{Prep}, implementing the offline phase in the $(\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Bootstrap}})$-hybrid model; and a novel way to authenticate bits to more than two parties, which takes as starting point the aBit command of [13], and which we model with the $\mathcal{F}_{\mathsf{Bootstrap}}$ functionality.

The online phase implements the standard functionality $\mathcal{F}_{\mathsf{Online}}$ (see Appendix C.2 for details). It is based on the $[\![\cdot]\!]$-representation of bits described in Section 2, and it is very similar to the online phase of other MPC protocols [6,7,8,13]. We compute a function represented as a binary circuit, where private inputs are additively shared among the parties, and correctness is guaranteed by using additive secret sharings of linear MACs with global secret key α. For simplicity we assume one single input for each party and one public output. The online protocol, presented in Section 5, uses the

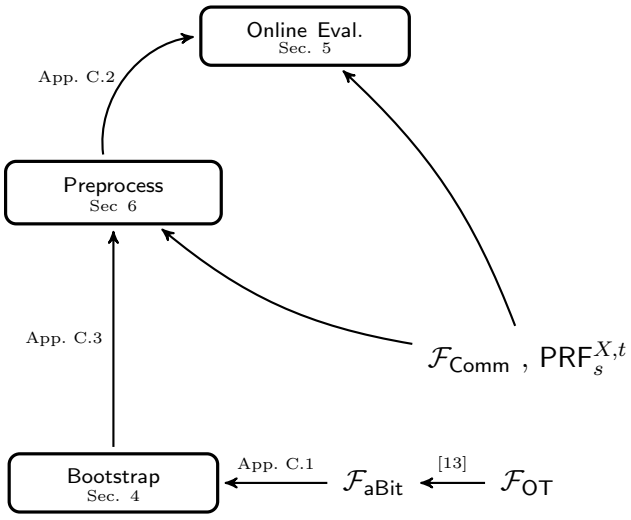

Figure 1 Overview of Protocols Enabling MPC

linearity of the $[\![\cdot]\!]$-sharings to perform additions and scalar multiplications locally. For general multiplications we need utilize data produced during the offline phase, in particular the output of the GaOT (Global authenticated OT) command of Section 6. Refer to Figure 2 for a complete description of the functionality for preprocessing data. The aforementioned command GaOT builds upon $\Pi_{\mathsf{Bootstrap}}$ protocol, described in Section 4, to generate random authenticated OTs and, as we noted above, we skip the less efficient procedures of [13].

The Functionality $\mathcal{F}_{\mathsf{Prep}}$

Let A be the set of indices of corrupt parties.

Initialize: On input (Init) from honest parties, the functionality samples random α_i for each $i \notin A$. It waits for the environment to input corrupt shares $\{\alpha_j\}_{j\in A}$ If any $j \in A$ outputs abort, then the functionality aborts and returns the set of $j \in A$ which returned abort. Otherwise the functionality sets $\alpha = \alpha_1 + \cdots + \alpha_n$, and outputs α_k to honest P_k.

Share: On input (i, x, Share) from party P_i, and (i, Share) from all other parties. The functionality produces an authentication $[\![x]\!] = (\langle x\rangle, \langle \mu\rangle)$. It sets $x_j = 0$ if $j \neq i$. Also, the MAC might be shifted by a value Δ_H, i.e. $\mu = x \cdot \alpha + \Delta_H$, where Δ_H is an $\mathbb{F}_2$-linear combination of $\{\alpha_k\}_{k\notin A}$ not known to the environment. It proceeds as follows:

- Set $\mu = x \cdot \alpha$. If $i \in A$, the environment specifies x.
- Wait for the environment to specify MAC shares $\{\mu_j\}_{j\in A}$, and generate $\langle \mu\rangle$ where the portion of honest shares is consistent with the adversarial shares, but otherwise random.
- Set $x_k = 0$ if $k \neq i$, $k \notin A$. If the environment inputs $\mathsf{shift\text{-}P_k}$ set $\mu_k = \mu_k + \alpha_k$.
- Output (x_k, μ_k) to honest P_k

GaOT: On input (GaOT) from the parties, the functionality waits for the environment to input "Abort" or "Continue". If it is told to abort, it outputs the special symbol $\varnothing$ to all parties.
Otherwise it samples three random bits e, x_0, x_1, and sets $z = x_e$. Then, for every bit $y \in \{e, z, x_0, x_1\}$ the functionality produces an authentication $[\![y]\!] = (\langle y\rangle, \langle \mu(y)\rangle)$, but let the environment to specify shares for corrupt P_j. It proceeds as follows:

- Set $\mu(y) = y \cdot \alpha$.
- Wait for the environment to input bit shares $\{y_j\}_{j\in A}$, and MAC shares $\{\mu_j\}_{j\in A}$, and creates sharings $\langle y\rangle$, $\langle \mu\rangle$ where the portion of honest shares is consistent with adversarial shares.
- Output (y_k, μ_k) to honest P_k.

Figure 2 Ideal Preprocessing

Notice that, as in [6,7,8,13], during the online computation of the circuit we do not know if we are working with the correct values, since we do not check the MACs of partially opened values during the computation. This check is postponed to the end of the protocol, where we call the MACCheck procedure as in [6] (see Appendix B for details). Note this procedure enables the checking of multiple sets of values partially opened during the computation without revealing the global secret key α, thus our MPC protocol can implement reactive functionalities.

The MAC checking protocol is called in both the offline and the online phases, it requires access to an ideal functionality for commitments $\mathcal{F}_{\mathsf{Comm}}$, also given in Appendix B, and it is not intended to implement any functionality. Also, note that the algebraic correctness of the output of the GaOT command in the offline phase is checked in the offline phase and not in the online phase.

4 From Tiny-OT aBit's to $[\![\cdot]\!]$-Sharings

At the heart of our MPC protocol is a method to translate from the two party aBits produced by the offline phase of the Tiny-OT protocol in [13], to the $[\![\cdot]\!]$-sharings under some global shared key

α from Section 2. We note that the protocol to produce aBit's is the only sub-protocol from [13] which we use in this paper, and thus the more complex protocols in [13] for producing aOT's and aAND's we discard. We first deal with the underlying two party sub-protocols, and then we use these to define our multi-party protocols.

4.1 Two-party $[\cdot]$-representations.

Thus throughout we assume access to an ideal functionality $\mathcal{F}_{\mathsf{aBit}}$, given in Figure 3, that produces a substantially unbounded number of (oblivious) authenticated *random* bits for two parties, under some *randomly* chosen key δ_j known by one of the parties. This functionality can be implemented assuming a functionality $\mathcal{F}_{\mathsf{OT}}$ and using OT-extension techniques as in [13]. For ease of exposition we present the functionality as returning single bits for single requests. In practice the functionality is implemented via OT-extension and so one is able to obtain *many* aBits on each invocation of the functionality, for a given value of δ_j. Adapting our protocols to deal with multiple aBit production for a single random fixed δ_j chosen by the functionality is left to the reader[4].

The Functionality $\mathcal{F}_{\mathsf{aBit}}$

Authenticated Bit(P_i, P_j): This functionality selects a random $\delta_j \in \mathbb{F}$ and a random bit r, and returns a sharing $[r]^i_{\delta_j,j}$.

- On input (aBit, i, j) from honest P_i and P_j, the functionality samples a random δ_j and a random sharing $[r]^i_{\delta_j,j} = (r, \mu_i, \nu_j)$, such that $\mu_i = \nu_j + r \cdot \delta_j$. It then outputs $\{r, \mu_i\}$ to P_i and $\{\delta_j, \nu_j\}$ to P_j.
- If P_i is corrupted, the functionality waits for the environment to input the pair $\{r, \mu_i\}$ and it sets $\nu_j = \mu_i + r \cdot \delta_j$ for some randomly chosen δ_j, and $\{\delta_j, \nu_j\}$ is returned to party P_j.
- If P_j is corrupted, the functionality waits for the environment to input the pair $\{\delta_j, \nu_j\}$, r is selected at random and μ_i is set to be $\nu_j - r \cdot \delta_j$. The pair $\{r, \mu_i\}$ is returned to party P_i.

Figure 3 Two-party Bit Authentication [13]

Using the protocol $\Pi_{\mathsf{2\text{-}Share}}$, described in Protocol 4, we can obtain a "two-party" representation $[r]^i_{\delta_j,j}$ of a random bit known to P_i, under the key *chosen* by P_j. This extension is needed because we need to adapt the aBit command to the multi-party case. For example, if two parties, P_i and P_j, run the command $\mathsf{aBit}(i, j)$, they obtain a random $[r]^i_{\delta'_j,j}$, with respect to δ'_j; when P_j calls $\mathsf{aBit}(k, j)$ with a different party $P_k, k \neq j$, then they obtain a random $[s]^k_{\tilde{\delta}_j,j}$, with a different $\tilde{\delta}_j$. Thus allowing the parties to select their own values of δ_j means that we can obtain key-consistent $[\cdot]$-representations, in which each party P_j use the same fixed δ_j. The security of the protocol $\Pi_{\mathsf{2\text{-}Share}}$ follows from the security of the original aBit in [13]: intuitively the changes required to obtain a consistent $[\cdot]$-representation do not compromise security, because δ_j is one-time-padded with the random δ'_j produced by $\mathcal{F}_{\mathsf{aBit}}$. See C.1 for details.

Notice that the command 2-Share takes δ_j as the input of P_j. In particular the value δ_j may not be used to authenticate bits. Thus we could use the protocol $\Pi_{\mathsf{2\text{-}Share}}$ to obtain a sharing of the *scalar product* $r \cdot \delta_j$, where P_i obtains the random bit r, and the other party decides what field element $\delta_j \in \mathbb{F}$ gets multiplied in. Then party P_i obtains the result μ_i masked by a one-time pad

[4] Note, that in this situation we (say) produce $1,000,000$ aBits per invocation with a fixed random value of δ_j, then on the next invocation we obtain another $1,000,000$ aBits but with a new random δ_j value. This is not explicit in the ideal functionality description of aBit presented in [13], but is implied by their protocol.

The Subprotocol $\Pi_{\mathsf{2\text{-}Share}}$

2Share$(i, j; \delta_j)$**:** On input $(\mathsf{2\text{-}Share}, i, j, \delta_i)$, where P_j has $\delta_j \in \mathbb{F}$ as input, this command produces a $[r]^i_{\delta_j, j}$ sharing of a random bit r.

1. P_i and P_j call $\mathcal{F}_{\mathsf{aBit}}$ on input (aBit, i, j): The box samples a random δ'_j and then produces

$$[r]^i_{\delta'_j, j} = (r, \mu'_i, \nu_j),$$

   such that $\mu'_i = \nu_j + r \cdot \delta'_j$, and outputs $\{r, \mu'_i\}$ to P_i and $\{\delta'_j, \nu_j\}$ to P_j.
2. P_j computes $\sigma_j = \delta_j + \delta'_j$ and sends σ_j to party P_i.
3. P_i sets $\mu_i = \mu'_i + r \cdot \sigma_j = \nu_j + r \cdot \delta_j$.

Protocol 4 Switching to Fixed δ-shares

value ν_j known only to P_j. This application of the subprotocol $\Pi_{\mathsf{2\text{-}Share}}$ is going to be crucial in our method to obtain authenticated OT's in our pre-processing phase. As a consequence we *do not always see* δ_j as an authentication key.

4.2 Multiparty $[\cdot]$-representation

The Functionality $\mathcal{F}_{\mathsf{Bootstrap}}$

Let A be the indices of corrupt parties.

Initialize: On input (Init) from honest parties, the functionality activates and waits for the environment to input a set of shares $\{\delta_j\}_{j \in A}$. It samples random $\delta \in \mathbb{F}$ and prepares sharing $\langle \delta \rangle$, where the portions of honest shares are consistent with the adversarial shares, but otherwise random. If any $j \in A$ outputs abort, then the functionality aborts and returns the set of $j \in A$ which returned abort, otherwise it continues.

Share: On input (i, x, Share) from party P_i, and (i, Share) from all other parties. The functionality produces a representation $[x]^i_\delta = (\langle x \rangle^i, \langle \mu \rangle^i, \langle \nu \rangle^{\mathcal{P}})$, except that ν might be shifted by a value Δ_H, i.e. $\mu = x \cdot \delta + \nu + \Delta_H$, where Δ_H is an $\mathbb{F}_2$-linear combination of $\{\delta_k\}_{k \notin A}$, which is not known to the environment. It proceeds as follows:

- It samples random $\mu \in \mathbb{F}$. If $i \in A$ waits for the environment to input (μ, x).
- The functionality sets $\nu = x \cdot \delta + \mu$.
- The functionality waits for the environment to input shares $\{\nu_j\}_{j \in A}$, and prepares sharing $\langle \nu \rangle^{\mathcal{P}}$ consistent with the adversarial shares. The portion of honest shares are otherwise random.
- If the environment inputs $\mathsf{shift\text{-}P_k}$, the functionality sets $\nu_k = \nu_k + \delta_k$, $k \notin A$.
- It outputs (ν_k, δ_k) to honest P_k.

Figure 5 Ideal Generation of $[\cdot]^i_{\delta, \mathcal{P}}$-representations

Here we show how to generalize the $\Pi_{\mathsf{2\text{-}Share}}$ protocol in order to obtain an n-party representation $[x]^i_\delta$ of a bit x chosen by P_i. This is what the functionality $\mathcal{F}_{\mathsf{Bootstrap}}$ models in Figure 5. It bootstraps from a two party authentication to a multi-party authentication of the shared bit. As before for $\Pi_{\mathsf{2\text{-}Share}}$, we can see the outputs of $\mathcal{F}_{\mathsf{Bootstrap}}$ as the shares of scalar products $x \cdot \delta$, where one party P_i chooses the scalar (bit) x, but now the field element δ is unknown and additively shared among all the parties. An interesting feature of this functionality is that the adversary can only influence *honest* outputs in a small way, that we model with the $\mathsf{shift\text{-}P_k}$ flag. Additionally, we can not prevent corrupt parties from outputting what they wish, this is reflected on the fact that the functionality leaves their outputs undefined. The main difference between this functionality and the equivalent in

the SPDZ protocol [7], is that in [7] the functionality takes as input an offset known to the adversary who adjusts his shares to obtain an invalid MAC value by this linear amount. We do not model this in our functionality, instead we allow the adversary to choose his shares arbitrarily (which obtains the same effect). However, in our protocol the adversary can also introduce an unknown (to the adversary) error into the MAC values. In particular the adversary can decide whether to shift honest shares, but he cannot choose the shifting, namely, an element on the $\mathbb{F}_2$-span of secrets δ_k of honest parties P_k. Later, we manage to determine whether there are any errors (both adversarially known and unknown ones) using an *information-theoretic* MACCheck procedure that we borrow from [6]. See Appendix B for details.

The protocol $\Pi_{\mathsf{Bootstrap}}$, described in Protocol 6, realizes the ideal functionality $\mathcal{F}_{\mathsf{Bootstrap}}$ in a hybrid model in which we are given access to $\mathcal{F}_{\mathsf{aBit}}$. It permits to obtain $[x]^i_\delta$ and it is implemented by sending to each $P_j, j \neq i$, a mask of x using the random bits given by **2-Share**$(i, j; \delta_j)$ as paddings, and then allowing P_j to adjust his share to the right value. In total the protocol needs to execute $n-1$ aBit per scalar product.

The Protocol $\Pi_{\mathsf{Bootstrap}}$

Initialize: Each party P_i samples a random δ_i. Define $\delta = \delta_1 + \cdots + \delta_n$.
Share: On input (i, x, Share) from P_i and (i, Share) from all other parties, do:

1. For each $j \neq i$, call $\Pi_{\text{2-Share}}$ with 2-Share$(i, j; \delta_j)$. Party P_i obtains $\{r_{i,j}, \mu_{i,j}\}_{j \neq i}$ whilst party P_j obtains $\nu_{i,j}$, such that $\mu_{i,j} = \nu_{i,j} + r_{i,j} \cdot \delta_j$.
2. Party P_i samples ϵ at random and sets $\mu_i = \epsilon + \sum_{j \neq i} \mu_{i,j}$ and $\nu_i = \epsilon + x \cdot \delta_i$.
3. Party P_i sends $d_j = x + r_{i,j}$ to party P_j for all $j \neq i$.
4. For $j \neq i$, P_j sets $\nu_j = \nu_{i,j} + d_j \cdot \delta_j$.
5. Output (μ_i, ν_i, δ_i) to P_i and (ν_j, δ_j) to party P_j, for $j \neq i$. The system now has $[x]^i_\delta$.

Protocol 6 Transforming Two-party Representations onto $[\cdot]^i_{\delta,\mathcal{P}}$-representations

Lemma 1. *In the $\mathcal{F}_{\mathsf{aBit}}$-hybrid model, the protocol $\Pi_{\mathsf{Bootstrap}}$ implements $\mathcal{F}_{\mathsf{Bootstrap}}$ with perfect security against any static adversary corrupting up to $n-1$ parties.*

Proof. See Appendix C.1.

5 The Online Phase

In this section we present the protocol Π_{Online}, described in Protocol 7, which implements the online functionality in the $(\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Prep}})$-hybrid model. The basic idea behind our online phase is to use the set of GaOTs output in the offline phase to evaluate each multiplication gate. To see how this is done, consider that we want to multiply two authenticated bits $[\![a]\!], [\![b]\!]$. The parties take a GaOT tuple $\{[\![e]\!], [\![z]\!], [\![x_0]\!], [\![x_1]\!]\}$ off the pre-computed list. Recall we have for such tuples $z = x_e$. It is then relatively straightforward to compute authenticated shares of $[\![c]\!]$, where $c = a \cdot b$, as follows: First, the parties partially open $[\![f]\!] = [\![b]\!] + [\![e]\!]$ and $[\![g]\!] = [\![x_0]\!] + [\![x_1]\!] + [\![a]\!]$, and then set $[\![c]\!] = [\![x_0]\!] + f \cdot [\![a]\!] + g \cdot [\![e]\!] + [\![z]\!]$. To see why this is correct, note that since, $x_e + x_0 + e \cdot (x_0 + x_1) = 0$, we have $c = x_0 + (b + e) \cdot a + (x_0 + x_1 + a) \cdot e + z = a \cdot b$.

Theorem 1. *In the $(\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Prep}})$-hybrid model, the protocol Π_{Online} securely implements $\mathcal{F}_{\mathsf{Online}}$ against any static adversary corrupting up to $n-1$ parties, assuming protocol* MACCheck *utilizes a secure pseudo-random function* $\mathsf{PRF}^{\mathbb{F},t}_s(\cdot)$.

Protocol Π_{Online}

Initialize: The parties call Init on the $\mathcal{F}_{\mathsf{Prep}}$ functionality to get the shares α_i of the global MAC key α. If $\mathcal{F}_{\mathsf{Prep}}$ aborts outputting a set of corrupted parties, then the protocol returns this subset of A. Otherwise the operations specified below are performed according to the circuit.

Input: To share his input bit x, P_i calls $\mathcal{F}_{\mathsf{Prep}}$ with input (i, x, Share) and party P_j for $i \neq j$ calls $\mathcal{F}_{\mathsf{Prep}}$ with input (i, Share). The parties obtain $[\![x]\!]$ where the x-share of P_j is set to zero if $j \neq i$.

Add: On input $([\![a]\!], [\![b]\!])$, the parties locally compute $[\![a+b]\!] = [\![a]\!] + [\![b]\!]$.

Multiply: On input $([\![a]\!], [\![b]\!])$, the parties call $\mathcal{F}_{\mathsf{Prep}}$ on input (GaOT), obtaining a random GaOT tuple $\{[\![e]\!], [\![z]\!], [\![x_0]\!], [\![x_1]\!]\}$. The parties then perform:

1. The parties locally compute $[\![f]\!] = [\![b]\!] + [\![e]\!]$ and $[\![g]\!] = [\![x_0]\!] + [\![x_1]\!] + [\![a]\!]$.
2. The shares $[\![f]\!]$ and $[\![g]\!]$ are partially opened.
3. The parties locally compute
$$[\![c]\!] = [\![x_0]\!] + f \cdot [\![a]\!] + g \cdot [\![e]\!] + [\![z]\!].$$

Output: This procedure is entered once the parties have finished the circuit evaluation, but still the final output $[\![y]\!]$ has not been opened.

1. The parties call the protocol Π_{MACCheck} on input of all the partially opened values so far. If it fails, they output $\varnothing$ and abort. $\varnothing$ represents the fact that the corrupted parties remain undetected in this case.
2. The parties partially open $[\![y]\!]$ and call Π_{MACCheck} on input y to verify its MAC. If the check fails, they output $\varnothing$ and abort, otherwise they accept y as a valid output.

Protocol 7 Secure Function Evaluation in the $\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Prep}}$-hybrid Model

Proof. See Appendix C.2.

6 The Offline Phase

Here we present our offline protocol Π_{Prep} (Protocol 8). The key part of this protocol is the GaOT command. In [13] the authors give a two-party protocol to enable one party, say A, to obtain two authenticated bits e, z, and the other party, say B, to obtain two authenticated secret bits x_0, x_1, such that $z = x_e$ and e, x_0 and x_1 are chosen at random. We generalize such a procedure to many parties and we obtain sharings $[\![e]\!]$, $[\![z]\!]$, $[\![x_0]\!]$, $[\![x_1]\!]$, subject to $z = x_e$. Notice that the values e, z, x_0, x_1 are not known so they can be used in the online phase to implement multiplication gates.

The idea behind the GaOT command it is to exploit the relation between "affine functions" and "selector functions", in which a bit e selects one of two elements (χ_0, χ_1) in $\mathbb{F}$. This connection was already noted in [1] on the context of garbling arithmetic circuits via randomized encodings. Thus, on one hand we have authentications, that are essentially evaluations of affine functions, and on the other we have OT quadruples, that can be seen as selectors. Seeing both as the same object means that a way to authenticate bits also gives us a way to generate OTs, and the other way around. The procedure is broken into three steps, **Share OT**, **Authenticate OT** and **Sacrifice OT**. We examine these three stages in turn.

To produce bit quadruples (e, z, x_0, x_1), such that $z = x_e$, the parties will use a (secret) affine line in $\mathbb{F}$ parametrized by (ϑ, η). Note that with our functionality $\mathcal{F}_{\mathsf{Bootstrap}}$ we get $[e_i]^i_\eta$, where e_i is known to P_i, and an additive sharing $\langle \eta \rangle$ is held by the system. We denote this concrete execution of the functionality as $\mathcal{F}_{\mathsf{Bootstrap}}(\eta)$, since we shall use fresh copies of $\mathcal{F}_{\mathsf{Bootstrap}}$ to generate more OT quadruples and also for authentication purposes. Note, that η is not an input to the functionality but a shared random value produced when initialising the functionality. Now, performing n independent queries of Share command on this copy $\mathcal{F}_{\mathsf{Bootstrap}}(\eta)$, the parties can generate

$$[e]^{\mathcal{P}}_\eta = [e_1]^1_\eta + \cdots + [e_n]^n_\eta. \tag{1}$$

The Protocol Π_{Prep}

Let A be the set of indices of corrupt parties.

Initialize: On input (Init) from honest parties and adversary, the system runs a copy of $\mathcal{F}_{\mathsf{Bootstrap}}$ which is denoted $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$. Then it calls Init on $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$. If $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ aborts, outputting a set of corrupted parties, then the protocol returns this subset of A and aborts. Otherwise, the values δ_i returned by $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ are labelled as α_i. Set $\alpha = \alpha_1 + \cdots + \alpha_n$, and output α_i to honest parties P_i.

Share: On input (i, x, Share) from party i and (j, Share) from all parties $j \neq i$. The protocol calls Share command of $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ to obtain $[x]^i_\alpha$, given by $\{\langle\mu\rangle^i, \langle\nu\rangle^{\mathcal{P}}\}$. Then, for $j \neq i$, party P_j sets his share of x to be zero, and $\mu_j(x) = \nu_j$. Party P_i sets $\mu_i(x) = \mu + \nu_i$. Thus, the parties obtain $[\![x]\!]$.

GaOT: On input (GaOT) from all P_i, execute the following sub-procedures:

Share OT. This generates sharings $(\langle e\rangle, \langle z\rangle, \langle x_0\rangle, \langle x_1\rangle)$ such that x_0, x_1 and e are random bits. If all parties are honest then it holds $z = x_e$.

1. The system runs a fresh copy of $\mathcal{F}_{\mathsf{Bootstrap}}$ on Init command getting an additive sharing $\langle\eta\rangle$ for some random $\eta \in \mathbb{F}$. Denote this copy as $\mathcal{F}_{\mathsf{Bootstrap}}(\eta)$.
2. Each party samples a random bit e_i. Define $e = e_1 + \cdots + e_n$.
3. For each $i = 1, \ldots, n$, the system calls $\mathcal{F}_{\mathsf{Bootstrap}}(\eta)$ on input (i, e_i, Share) from party P_i and input (i, Share) from any other P_j, to obtain $[e_i]^i_\eta$. That is, (in an honest execution) P_i gets $\zeta_i \in \mathbb{F}$, and the parties gets an additive sharing $\langle\vartheta_i\rangle$ of some unknown $\vartheta_i \in \mathbb{F}$, such that $\zeta_i = \vartheta_i + e_i \cdot \eta$. The parties compute $[e]^{\mathcal{P}}_\eta = [e_1]^1_\eta + \cdots + [e_n]^n_\eta$.
4. At this point of the protocol, the system holds sharings $\langle e\rangle$, $\langle\zeta\rangle$, $\langle\vartheta\rangle$, $\langle\eta\rangle$, so it can derive $\langle\chi_0\rangle = \langle\vartheta\rangle$, and $\langle\chi_1\rangle = \langle\vartheta\rangle + \langle\eta\rangle$. Note that (for an honest execution) $\zeta = \vartheta + e \cdot \eta$, or in other words $\zeta = \chi_e$.
5. Each party P_i sets z_i, $x_{0,i}$, $x_{1,i}$ to be the least significant bits of ζ_i, $\chi_{0,i}$, $\chi_{1,i}$ respectively, so as to obtain sharings $\langle z\rangle, \langle x_0\rangle$ and $\langle x_1\rangle$.

Authenticate OT. This step produces authentications on the bits previously computed. For every bit $y \in \{e, z, x_0, x_1\}$ it does the following:

6. Call $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ on input (i, y_i, Share) from P_i and (j, Share) for party P_j to obtain $[y_i]^i_\alpha$.
7. Compute $[\![y]\!]$ by forming $\sum_{i\in\mathcal{P}}[y_i]^i_\alpha$, and then subtracting $\nu(y)$ from $\mu(y)$.

Sacrifice OT. This step checks that the authenticated OT-quadruples are correct. Let $[\![e]\!], [\![z]\!], [\![x_0]\!], [\![x_1]\!]$, be the quadruple to check, and κ a security parameter:

8. Every party P_i samples a seed s_i and asks $\mathcal{F}_{\mathsf{Comm}}$ to broadcast $\tau_i = \mathsf{Comm}(s_i)$.
9. Every party P_i calls $\mathcal{F}_{\mathsf{Comm}}$ with $\mathsf{Open}(\tau_i)$ and all parties obtain s_j for all j. Set $s = s_1 + \cdots + s_n$.
10. Parties sample a random vector $\mathbf{t} = \mathsf{PRF}^{\mathbb{F}_2,\kappa}_s(0) \in \mathbb{F}_2^\kappa$. Note all parties obtain the same vector as they have agreed on the seed s.
11. For $i = 1, \ldots, \kappa$, repeat the following:
    - Take one fresh quadruple $[\![e_i]\!], [\![z_i]\!], [\![x_{0,i}]\!], [\![x_{1,i}]\!]$, and partially open the values $p_i = t_i \cdot ([\![x_0]\!] + [\![x_1]\!]) + [\![x_{0,i}]\!] + [\![x_{1,i}]\!]$ and $q_i = [\![e]\!] + [\![e_i]\!]$.
    - Locally evaluate c_i such that

$$[\![c_i]\!] = t_i \cdot ([\![z]\!] + [\![x_0]\!]) + [\![z_i]\!] + [\![x_{0,i}]\!] + p_i \cdot [\![e]\!] + q_i \cdot ([\![x_{0,i}]\!] + [\![x_{1,i}]\!]),$$

      and check it partially opens to zero. If it does not, then abort.
12. The parties call Π_{MACCheck} on the values partially opened in step 11.
13. If no abort occurs, output $[\![e]\!], [\![z]\!], [\![x_0]\!], [\![x_1]\!]$ as a valid quadruple.

Protocol 8 Preprocessing: Input Sharing and Creation of OT Quadruples in the $\mathcal{F}_{\mathsf{Bootstrap}}$-hybrid Model

Thus, the system obtains two (secret) elements $\langle e\rangle$, $\langle\zeta\rangle$, such that $\zeta = \vartheta + e \cdot \eta$, for line $(\langle\vartheta\rangle, \langle\eta\rangle)$. Define $\chi_0 = \vartheta$ and $\chi_1 = \vartheta + \eta$, so it holds $\zeta = \chi_e$. The quadruple (e, z, x_0, x_1) is then given by the least significant bits of the corresponding field elements $(e, \zeta, \chi_0, \chi_1)$. This conclude the **Share OT** step.

To add MACs to each bit of the quadruple that the parties just generated, the protocol uses the $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ instance to obtain a sharing $\langle\alpha\rangle$ of the global key. Each party can now authenticate his shares of (e, z, x_0, x_1) querying Share command and obtaining $[\![e]\!]$, $[\![z]\!]$, $[\![x_0]\!]$, $[\![x_1]\!]$. We emphasize

that the same α is used to authenticate all OT quadruples, thus $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ is fixed once and for all.

After the **Authenticate OT** step the parties have sharings $[\![e]\!]$, $[\![z]\!]$, $[\![x_0]\!], [\![x_1]\!]$, which could suffer from two possible errors induced by the corrupted parties: Firstly the algebraic equation $z = x_e$ may not hold, and second the MAC values may be inconsistent. For the latter problem we will check all the partially opened values using the MACCheck procedure at the end of the offline phase. For the former case we use the **Sacrifice OT** step. We use the same methodology as in [4,7,6], i.e. one quadruple is checked by "sacrificing" another quadruple. The idea involving sacrificing can be seen as follows: We associate to each pair of quadruples a polynomial $S(t)$ over the field of secrets ($\mathbb{F}_2$ in our case), which is the zero polynomial only if both quadruples are correct. Thus, proving correctness of quadruples is equivalent to proving that $S(t)$ is the zero polynomial. This is done by securely evaluating $S(t)$ on a random public challenge bit t via a combination of addition gates and two openings (plus one extra opening to check the evaluation), and then checking that the result of the evaluation partially opens to zero. In this way we would waste κ quadruples to check one quadruple, to get security of $2^{-\kappa}$; we refer the reader to Appendix A for a more efficient sacrifice procedure.

Theorem 2. *Let κ be the security parameter and $t \in \mathbb{N}$. In the $(\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Bootstrap}})$-hybrid model, the protocol Π_{Prep} securely implements $\mathcal{F}_{\mathsf{Prep}}$ with statistical security on κ against any static adversary corrupting up to $n-1$ parties, assuming the existence of $\mathsf{PRF}_s^{X,m}(\cdot)$ with domain $X = \mathbb{F}$ (resp. $\mathbb{F}_2$) and $m = t$ (resp. κ).*

Proof. See Appendix C.3

7 Efficiency Analysis

As it stands our protocol is not that efficient, mainly due to the naive sacrificing step performed in the offline phase so as to check the GaOTs for correctness. In Appendix A we present a much more efficient sacrifice step, which for reasonable parameters means that the ratio of required GaOT's for each used one can be between four and six. Let this ratio be denoted r.

We examine the cost of a multiplication in terms of the number of aBits required in the case of two parties. We notice that each GaOT requires us to consume ten aBits; we need to execute the **Share OT** step to determine e, z, x_0, x_1 (which requires one aBit consumption per player, i.e. two in total when $n = 2$); in addition each of these four bits needs to be authenticated in **Authenticate OT** in Protocol 8 (which again requires one aBit consumption per player, i.e. eight in total when $n = 2$).

Since we need one checked GaOT to perform a secure multiplication, and we sacrifice $r-1$ GaOT to obtain a checked one; this means we require $r \cdot 10$ aBits per secure multiplication in the two party case. Depending on the parameters we use for our sacrifice step in Appendix A, this equates to 40, 50 or 60 aBits per secure multiplication.

We now compare this to the number of aBits needed in the Tiny-OT protocol [13]. In this protocol each secure multiplication requires two aBits, two aANDs and two aOTs. Assuming a bucket size of four in the protocols to generate aANDs and aOTs; each aAND (resp. aOT) requires four LaANDs (resp LaOTs). Each LaAND requires four aBits and each LaOT requires three aBits. Thus the total number of aBits per secure multiplication is $2 \cdot (1 + 4 \cdot 4 + 4 \cdot 3) = 2 \cdot 29 = 58$. We see therefore that we can make our protocol (in the two party case) more efficient than the Tiny-OT protocol, when we measure efficiency in terms of the number of aBits consumed.

8 Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant EP/I03126X and by research sponsored by Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

1. B. Applebaum, Y. Ishai, and E. Kushilevitz. How to garble arithmetic circuits. In R. Ostrovsky, editor, *FOCS*, pages 120–129. IEEE, 2011.
2. G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM Conference on Computer and Communications Security*, pages 535–548. ACM, 2013.
3. D. Beaver. Efficient multiparty protocols using circuit randomization. In J. Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.
4. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.
5. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.
6. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure mpc for dishonest majority - or: Breaking the spdz limits. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.
7. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [15], pages 643–662.
8. I. Damgård and S. Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In A. Sahai, editor, *TCC*, volume 7785 of *Lecture Notes in Computer Science*, pages 621–641. Springer, 2013.
9. T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 537–556. Springer, 2013.
10. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In A. V. Aho, editor, *STOC*, pages 218–229. ACM, 1987.
11. D. Harnik, Y. Ishai, and E. Kushilevitz. How many oblivious transfers are needed for secure multiparty computation? In A. Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 284–302. Springer, 2007.
12. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In D. Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.
13. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [15], pages 681–700.
14. J. B. Nielsen and C. Orlandi. Lego for two-party secure computation. In *TCC*, pages 368–386, 2009.
15. R. Safavi-Naini and R. Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.

A Batching the Sacrifice Step

This technique (an adaptation of a technique to be found originally in [14,6,9]) permits to check a batch of OT quadruples for algebraic correctness using a *smaller* number of "sacrificed" quadruples

than the basic version we described in Section 6. Recall, the idea is to check that an authenticated OT-quadruple $\mathsf{GaOT}_i = (\llbracket e_i \rrbracket, \llbracket z_i \rrbracket, \llbracket x_i \rrbracket, \llbracket y_i \rrbracket)$ verifies the "multiplicative" relation $m_i = z_i + x_i + e_i \cdot (x_i + y_i) = 0$.

At a high level, Protocol 9 essentially consists of two different phases. Let $(\mathsf{GaOT}_1, \ldots, \mathsf{GaOT}_N)$ be a set of OT quadruples, in the first phase a fixed portion of these GaOTs are partially opened as in a classical cut-and-choose step. If any of the opened OT quadruples does not satisfy the multiplicative relation the protocol aborts. Otherwise it runs the second phase: the remaining GaOTs are permuted and uniformly distributed into t buckets of size T. Then, for each of the buckets, the protocol selects a $\mathsf{BucketHead}$, i.e. the first (in the lex order) GaOT in the bucket (as in [9]), and uses the remaining GaOTs in the same bucket to check that $\mathsf{BucketHead}$ correctly satisfies the multiplicative relation. If any $\mathsf{BucketHead}$ does not pass the test, then we know that some parties are corrupted and the protocol aborts. If all the checks pass then we obtain t algebraically correct $\mathsf{BucketHead}$s, i.e. t OT quadruples, with overwhelming probability.

Bucket Cut-and-Choose Protocol

Input : Let $N = (T + h) \cdot t$ be the number of input GaOTs and T the size of the buckets, with $T \geq 2$. We let $1 \leq h \leq T$ denote an additional parameter controlling how much cut-and-choose we perform.

Phase-I *Cut-And-Choose* :

1. Every party P_i samples a seed s_i and asks $\mathcal{F}_{\mathsf{Comm}}$ to broadcast $\tau_i = \mathsf{Comm}(s_i)$.
2. Every party P_i calls $\mathcal{F}_{\mathsf{Comm}}$ with $\mathsf{Open}(\tau_i)$ and all parties obtain s_j for all j. Set $s = s_1 + \cdots + s_n$.
3. Using a $\mathsf{PRF}_s^{\mathbb{F}_2, N}$, parties sample a random vector $\mathbf{v} \in \mathbb{F}_2^N$, such that the number of its non-zero entries is $h \cdot t$ (i.e. the Hamming weight of $\mathbf{v}$ is $h \cdot t$).
4. Let $\mathcal{J}$ be the set of indices j such that $v_j \neq 0$, and, $\forall j \in \mathcal{J}$, the parties partially open GaOT_j and check that it satisfies the algebraic relation $z_j + x_j = e_j \cdot (x_j + y_j)$. If there exists an algebraically incorrect GaOT_j quadruple, then the protocol aborts.

Phase-II *Bucket-Sacrifice* :

5. Permute the unopened GaOTs according to a random permutation π on $T \cdot t$ indices, again using a PRF_s. Then renumber the permuted unopened GaOT_j, such that $j = 1, \ldots, T \cdot t$, and, for $i = 1, \ldots, t$, create the ith bucket as $\{\mathsf{GaOT}_j\}_{j=iT-T+1}^{iT}$.
6. Parties compute a $\mathsf{BucketHead}(i)$ for each $i = 1, \ldots, t$, i.e. return the first (in the lex order) element in the ith bucket.
7. For $i = 1, \ldots, t$, parties check that $\mathsf{BucketHead}(i) = \mathsf{GaOT}_i = (\llbracket e_i \rrbracket, \llbracket z_i \rrbracket, \llbracket x_i \rrbracket, \llbracket y_i \rrbracket)$ is correct using the other GaOTs in the bucket. For $j = iT - T + 2, \ldots, iT$ do:
   - Set $\mathsf{CheckGaOT}_j = \mathsf{GaOT}_j = (\llbracket \mathfrak{z}_j \rrbracket, \llbracket \mathfrak{h}_j \rrbracket, \llbracket \mathfrak{e}_j \rrbracket, \llbracket g_j \rrbracket)$.
   - Parties open $\langle e_i + \mathfrak{e}_j \rangle$ and $\langle x_i + y_i + \mathfrak{h}_j + \mathfrak{g}_j \rangle$.
   - Parties locally compute

$$\llbracket c_{i,j} \rrbracket = \llbracket z_i + x_i \rrbracket + \llbracket \mathfrak{z}_j + \mathfrak{h}_j \rrbracket + (e_i + \mathfrak{e}_j) \llbracket \mathfrak{h}_j + \mathfrak{g}_j \rrbracket + (x_i + y_i + \mathfrak{h}_j + \mathfrak{g}_h) \llbracket e_i \rrbracket,$$

     and check it partially opens to zero.
   - If all checks go through output GaOT_i as valid quadruples; otherwise abort.
8. The parties execute the protocol Π_{MACCheck} to check all partially opened values.

Protocol 9 Bucket Cut-and-Choose Protocol

Theorem 3. *For $T \geq \frac{\kappa + \log_2(t)}{\log_2(t)}$ the previous protocol provide t correct GaOTs with error probability $2^{-\kappa}$.*

Proof (sketch). It is easy to check that the protocol is correct and secure in the semi-honest model, i.e. if all the OT quadruples are honestly generated, according to the GaOT command in Π_{Prep}, then $c_i = 0, \forall i$.

The argument for active security is as follows. A $\mathsf{badGaOT}$, i.e. a OT quadruple which does not satisfy the multiplicative relation, passes the test if and only if all the partially opened GaOTs in the cut-and-choose phase are correct and then it ends up in a bucket containing only $\mathsf{badGaOT}$s. This is because if we combine two $\mathsf{badGaOT}$s, say GaOT_i and GaOT_j, we obtain $c_{i,j} = m_i + m_j = 1+1 = 0$, and the test passes. We show that this happens with negligible probability with an appropriate choice of the parameters. We argue this in two steps: first we prove that when a bucket contains at least one $\mathsf{goodGaOT}$ (a OT satisfying the multiplicative relation) a $\mathsf{badGaOT}$ will be always detected, and then we bound the probability of having buckets containing only $\mathsf{badGaOT}$s.

If parties misbehaved in any previous step yielding a $\mathsf{badGaOT}_i$, when we combine it with a $\mathsf{goodGaOT}_j$, then $c_{i,j} = m_i + m_j = 1$ and the check fails. Notice that the protocol always abort if there is a bucket with both bad and good GaOTs. More precisely the protocol checks the algebraic correctness of the $\mathsf{BucketHead}$s, but indirectly also that of any other GaOTs (We use the $\mathsf{BucketHead}$ notation so that each GaOT is only once paired with a different GaOT).
Let

- $\mathsf{PassICheck}$ be the event that the protocol does not abort in the cut-and-choose step
- $m\mathsf{badGaOT}$ be the event that m GaOTs are bad. Note that we fix m here.
- $\mathsf{NoMixedBucket}$ the event that there are no buckets containing both $\mathsf{goodGaOT}$s and $\mathsf{badGaOT}$s.

We bound the probability that both $\mathsf{PassICheck}$ and $\mathsf{NoMixedBucket}$ occur. To do this we prove:

1. $\Pr[E_1] = \Pr[\mathsf{PassICheck} \wedge m\mathsf{badGaOT}] \leq (\frac{T}{T+h})^m$.
2. $\Pr[E_2] = \Pr[E_1 \wedge \mathsf{NoMixedBucket}] \leq 2^{(\log_2(t))(1-T)}$.

The first point is straightforward. First note that if $m > h \cdot t$ then $\Pr[\mathsf{PassICheck}] = 0$ and the protocol aborts; similarly, if $m < T$, then $\Pr[\mathsf{NoMixedBucket}] = 0$, so we can suppose $T \leq m \leq h \cdot t$ (in particular $m > 1$). Moreover as a bad $\mathsf{BucketHead}$ will be always detected if a bucket contains both good and bad GaOTs, we add the condition $m = k \cdot T$, $k = 1, \ldots, t$. In this way if m denotes the number of $\mathsf{badGaOT}$s, and $\mathsf{PassICheck}$ is true, then the $h \cdot t$ GaOTs that are opened in the cut-and-choose step are sampled from the $N - m$ good GaOTs. It holds:

$$\begin{aligned}
\Pr[E_1] = & \binom{N-m}{h\cdot t} \cdot \binom{N}{h\cdot t}^{-1} = \binom{(T+h)\cdot t - m}{T\cdot t - m} \cdot \binom{(T+h)\cdot t}{T\cdot t}^{-1} = \\
= & \frac{((T+h)\cdot t - m)\cdots(h\cdot t+1)\cdot(Tt)!}{(T\cdot t + h\cdot t)\cdots(ht+1)(Tt-m)!} = \frac{(T\cdot t)\cdots(Tt-m+1)\cdot(Tt-m)!}{(Tt+ht)\cdots(Tt+ht-m+1)\cdot(Tt-m)!} \leq \\
\leq & \Big(\frac{Tt}{Tt+ht}\Big)^m = \Big(\frac{T}{T+h}\Big)^m.
\end{aligned}$$

Now we compute the probability of $\mathsf{NoMixedBucket} \wedge E_1$. Recall that the cardinality of each of the t buckets is T and that we are assuming $m = k \cdot T$ bad GaOTs. It is easy to see that

$$\Pr[E_2] \leq \Big(\frac{T}{T+h}\Big)^{kT} \cdot \binom{t}{k} \cdot \binom{Tt}{k\cdot T}^{-1}.$$

This probability is maximized in $k = 1$. Intuitively we can see this as follows: the term $\binom{t}{k} \cdot \binom{Tt}{k\cdot T}^{-1}$ is symmetric with respect to the value $k = t/2$, as $\binom{t}{k} \cdot \binom{Tt}{k\cdot T}^{-1} = \binom{t}{t-k} \cdot \binom{Tt}{Tt-k\cdot T}^{-1}$, $k = 1, \ldots, t$,

and it strictly decreases for $1 \leq k \leq t/2$; the term $\left(\frac{T}{T+h}\right)^{kT}$ is less than 1 and it decreases when k grows. So when we multiply the two terms we have that the above probability for values of k in $[1, \ldots, t/2]$ is bigger than the same probability for "symmetric" values in $]t/2, \ldots, t]$ and we have the maximum for $k = 1$. By substituting this value in the previous expression we get:

$$\begin{aligned}\Pr[E_2] &= \left(\frac{T}{T+h}\right)^{T} \cdot t \cdot \binom{Tt}{T}^{-1} \\ &\leq \left(\frac{T}{T+h}\right)^{T} \cdot t^{(1-T)} = 2^{(\log_2(t))(1-T)+T(\log_2(T/(T+h))} \\ &\leq 2^{(\log_2(t))(1-T)}\end{aligned}$$

Thus for $T \geq \frac{\kappa+\log_2(t)}{\log_2(t)}$ we obtain $\Pr[E_2] \leq 2^{-\kappa}$.

□

We can replace the **Sacrifice OT** step in Π_{Prep} with the above Bucket-Cut-and-Choose Protocol and Theorem 2, with relative proof, still holds.

Notice, how the value h has little effect on the final probability (we suppressed the effect in the statement of the Lemma since it is so low). This means we can take $h = 1$ to obtain the most efficient protocol, which means the amount of cut-and-choose performed is relatively low.

To measure the efficiency of this protocol we can consider the ratio $r = \frac{(T+h)\cdot t}{t} = T + h$: it measures the number of GaOTs that we need to produce one actively secure OT quadruple. We obtain the following table, all with $h = 1$ and an error probability of 2^{-40}.

r	$T = r - h$	t	$\frac{40+\log_2(t)}{\log_2(t)}$
4	3	2^{20}	3
5	4	2^{14}	3.85
6	5	2^{10}	5

B Information Theoretic Tags for Dishonest Majority

In the online phase, parties work with representations with *information-theoretic* message authentication codes. The key properties of the MACs is that are homomorphic, and hold enough entropy to convince an honest party that local computation has been done correctly. The homomorphic property allows us to postpone the check of the correctness in the MACs until the very end of the circuit evaluation (where the circuit can be the one implicitly used in the preprocessing or the target online circuit). In [6] it was shown how to do the check on partially open values whilst keeping secret the key, hence enabling support for reactive online evaluations, and this is the one we use. See Protocol 10 for details. The procedure utilizes an ideal functionality $\mathcal{F}_{\mathsf{Comm}}$ for commitments given in Figure 11. An implementation of $\mathcal{F}_{\mathsf{Comm}}$ in the random oracle model can be found in the Appendix of [6].

In order to understand the probability of an adversary being able to cheat during the execution of Protocol 10, the authors in [6] used a security game approach, which in turn was an adaptation of the one in [7]. For completeness, we state here both the protocol and the security game.

The adversary wins the game if there is an $i \in \{1, \ldots, t\}$ for which $b_i \neq a_i$, and the check goes through. The second step in the game, where the adversary sends the b_i's, models the fact that

Protocol Π_{MACCheck}

Usage: The parties have a set of $[\![a_i]\!]$, sharings and public bits b_i, for $i = 1, \ldots, t$, and they wish to check that $a_i = b_i$, i.e. they want to check whether the public values are consistent with the shared MACs held by the parties.

As input the system has sharings $\big(\langle\alpha\rangle, \{b_i, \langle a_i\rangle, \langle\mu(a_i)\rangle\}_{i=1}^{t}\big)$. If the MAC values are correct then we have that $\mu(a_i) = b_i \cdot \alpha$, for all i.

$\mathsf{MACCheck}(\{b_1, \ldots, b_t\})$**:**

1. Every party P_i samples a seed s_i and asks $\mathcal{F}_{\mathsf{Comm}}$ to broadcast $\tau_i = \mathsf{Comm}(s_i)$.
2. Every party P_i calls $\mathcal{F}_{\mathsf{Comm}}$ with $\mathsf{Open}(\tau_i)$ and all parties obtain s_j for all j.
3. Set $s = s_1 + \cdots + s_n$.
4. Parties sample a random vector $\boldsymbol{\chi} = \mathsf{PRF}_s^{\mathbb{F},t}(0) \in \mathbb{F}^t$; note all parties obtain the same vector as they have agreed on the seed s.
5. Each party computes the public value $b = \sum_{i=1}^{t} \chi_i \cdot b_i \in \mathbb{F}$.
6. The parties locally compute the sharings $\langle\mu(a)\rangle = \chi_1 \cdot \langle\mu(a_1)\rangle + \cdots + \chi_t \cdot \langle\mu(a_t)\rangle$ and $\langle\sigma\rangle = \langle\mu(a)\rangle - b \cdot \langle\alpha\rangle$.
7. Party i asks $\mathcal{F}_{\mathsf{Comm}}$ to broadcast his share $\tau_i' = \mathsf{Comm}(\sigma_i)$.
8. Every party calls $\mathcal{F}_{\mathsf{Comm}}$ with $\mathsf{Open}(\tau_i')$, and all parties obtain σ_j for all j.
9. If $\sigma_1 + \cdots + \sigma_n \neq 0$, the parties output $\varnothing$ and abort, otherwise they accept all b_i as valid authenticated bits.

Protocol 10 Method to Check MACs on Partially Opened Values

Game: Security of the MACCheck procedure assuming pseudorandom functions

1: The challenger samples random sharing $\langle\alpha\rangle \in \mathbb{F}$. It sets $\langle\mu(a_i)\rangle = a_i \cdot \langle\alpha\rangle$ and sends bits $a_1, \ldots, a_t$ to the adversary.
2: The adversary sends back bits $b_1, \ldots, b_t$.
3: The challenger generates random values $\chi_1, \ldots, \chi_t \in \mathbb{F}$ and sends them to the adversary.
4: The adversary provides an error $\Delta \in \mathbb{F}$.
5: Set $b = \sum_{i=1}^{t} \chi_i \cdot b_i$, and sharings $\langle\mu(a)\rangle = \sum_{i=0}^{t} \chi_i \cdot \langle\mu(a_i)\rangle$, and $\langle\sigma\rangle = \langle\mu(a)\rangle - b \cdot \langle\alpha\rangle$. The challenger checks that $\sigma = \Delta$.

corrupted parties can choose to lie about their shares of values opened on the execution of the parent protocol. The offset Δ models the fact that the adversary is allowed to introduce errors on the MACs. A formal proof of Theorem 4 can be found in the Appendix of [7,6].

Theorem 4 ([6]). *The protocol* $\mathsf{MACCheck}$ *is correct, i.e. it accepts if all the public values* b_i*, and the corresponding MACs are correctly computed. Moreover, it is sound, i.e. it rejects except with probability* $\frac{2}{|\mathbb{F}|}$ *in case at least one value, or MAC, is not correctly computed.*

The Functionality $\mathcal{F}_{\mathsf{Comm}}$

Commit: On input $(\mathsf{Comm}, v, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), where v is either in a specific domain or $\perp$, it stores (v, i, τ_v) on a list and outputs (i, τ_v) to all parties and adversary.

Open: On input $(\mathsf{Open}, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), the ideal functionality outputs (v, i, τ_v) to all parties and adversary. If $(\mathsf{NoOpen}, i, \tau_v)$ is given by the adversary, and P_i is corrupt, the functionality outputs $(\perp, i, \tau_v)$ to all parties.

Figure 11 Ideal Commitments

C Security Proofs

C.1 Proof of the Bootstrap Step (Lemma 1)

We show that an environment $\mathcal{Z}$ corrupting up to $n-1$ parties, playing with $\Pi_{\mathsf{Bootstrap}}$ attached to $\mathcal{F}_{\mathsf{aBit}}$ or with the simulator $\mathcal{S}$ attached to $\mathcal{F}_{\mathsf{Bootstrap}}$, sees transcripts that are identically distributed. We assume *authenticated* communication between parties, that is, they are given access to a functionality $\mathcal{F}_{\mathsf{AT}}$, which on input (m, s, s') from P_s, it gives message m to $P_{s'}$ and also leak it to $\mathcal{Z}$. In a nutshell, the simulator runs a copy of $\Pi_{\mathsf{Bootstrap}}$ acting on behalf of honest parties. Let A be the set of indices of corrupted parties, parties in A are indexed with j, and parties not in A with k.

We start describing the behaviour of $\mathcal{S}$. The corruption is static, so we can distinguish the two cases:

a) P_i **is honest.**
   1. In step 1, for $s \in \mathcal{P}$, $\mathcal{S}$ engages in a run of $\Pi_{\mathsf{2\text{-}Share}}(\mathsf{2\text{-}Share}, i, s)$ with $\mathcal{Z}$, acting on behalf of P_i and honest P_k: It sets an internal copy of $\mathcal{F}_{\mathsf{aBit}}$ to generate representations $[r_{i,j}]^i_{\delta'_j}$ on dummy bits $r_{i,j}$. It answers queries from $\mathcal{Z}$ by sending him $\{\nu_{i,j},\, \delta'_j\}_{j\in A}$. $\mathcal{S}$ also gives random σ_k to $\mathcal{Z}$, for $k \notin A$, and gets back σ^*_j for $j \in A$ (acting as $\mathcal{F}_{\mathsf{AT}}$). It then sets $\delta^*_j = \sigma^*_j + \delta'_j$.
   2. $\mathcal{S}$ sends $\{\delta^*_j\}_{j\in A}$ to $\mathcal{F}_{\mathsf{Bootstrap}}$ as part of **Initialize**.
   3. In step 3, $\mathcal{S}$ acting as $\mathcal{F}_{\mathsf{AT}}$ gives random d_s to $\mathcal{Z}$, $\forall s \neq i$. Note that $\nu^*_j = \nu_{i,j} + d_j \cdot \delta^*_j$ is the purported share that corrupt P_j should come up with.
   4. $\mathcal{S}$ sends $\{\nu^*_j\}_{j\in A}$ to $\mathcal{F}_{\mathsf{Bootstrap}}$.

b) P_i **is dishonest** ($\mathcal{Z}$ specifies input bit x).
   1. In step 1, for $s \in \mathcal{P}$, $\mathcal{S}$ engages in a run of $\Pi_{\mathsf{2\text{-}Share}}(\mathsf{2\text{-}Share}, i, s)$ with $\mathcal{Z}$, acting on behalf of honest P_k. It sets an internal copy of $\mathcal{F}_{\mathsf{aBit}}$ to generate representations $[r_{i,j}]^i_{\delta'_j}$ on dummy bits $r_{i,s}$, for $s \neq i$. $\mathcal{S}$ answers queries from $\mathcal{Z}$ by sending him $\{r_{i,s}, \mu'_{i,s}\}_{s\in\mathcal{P}}$, and $\{\nu_{i,j},\, \delta'_j\}_{j\in A}$. Acting as $\mathcal{F}_{\mathsf{AT}}$, $\mathcal{S}$ gives random σ_k to $\mathcal{Z}$ and it gets back σ^*_j. It then sets corrupt $\delta^*_j = \sigma^*_j + \delta'_j$. $\mathcal{S}$ also extracts $\nu^*_j = \nu_{i,j} + (x + r_{i,j}) \cdot \delta^*_j$, for $j \in A$, and $\mu_i = \sum_{s\neq i} \mu_{i,s}$ and $\nu^*_i = x \cdot \delta^*_i$.
   2. $\mathcal{S}$ sends $\{\delta^*_j\}_{j\in A}$ to $\mathcal{F}_{\mathsf{Bootstrap}}$ as part of **Initialize**.
   3. In step 3, $\mathcal{S}$ gets bits d^*_s for $s \neq i$ via $\mathcal{F}_{\mathsf{AT}}$, and for each $k \notin A$ sets the flag $\mathsf{shift\text{-}P_k}$ to true if $d^*_k \neq r_{i,k} + x$.
   4. $\mathcal{S}$ sends $\{\mathsf{shift\text{-}P_k}\}_{k\notin A}$, $\{\nu^*_j\}_{j\in A}$, μ_i, x, to $\mathcal{F}_{\mathsf{Bootstrap}}$.

Case honest P_i. First, we show that $\Pi_{\mathsf{Bootstrap}}$ and $\mathcal{F}_{\mathsf{Bootstrap}}$ output identically distributed values if $\mathcal{Z}$ is honest-but-curious. In $\Pi_{\mathsf{Bootstrap}}$, the parties obtains a sharing $\langle\delta\rangle$, $\langle\nu\rangle$, and party P_i provides input bit x and also obtains a field element μ. Then, we have

$$
\begin{aligned}
\sum_{s\in\mathcal{P}} \nu_s + x\cdot\delta &= \big(\epsilon + x\cdot\delta_i\big) + \Big(\sum_{s\neq i}\big(\nu_{i,s} + d_s\cdot\delta_s\big)\Big) + x\cdot\delta,\\
&= \big(\epsilon + x\cdot\delta_i\big) + \Big(\sum_{s\neq i}\big((\mu_{i,s} + r_{i,s}\cdot\delta_s) + (x + r_{i,s})\cdot\delta_s\big)\Big) + x\cdot\delta,\\
&= \big(\epsilon + x\cdot\delta_i\big) + \Big(\sum_{s\neq i}\big(\mu_{i,s} + x\cdot\delta_s\big)\Big) + x\cdot\delta,\\
&= \epsilon + \sum_{s\neq i}\mu_{i,s},\\
&= \mu.
\end{aligned}
$$

For what $\mathcal{Z}$ sees during the execution, either σ_k or d_s, leaked by $\mathcal{F}_{\mathsf{AT}}$, look random since they are paddings of δ_k and x, with fresh pads δ'_k, $r_{i,s}$ given by $\mathcal{F}_{\mathsf{aBit}}$ to P_i. Now, denote by δ_H the sum of the portion of δ-shares that honest parties generated in **Initialize** of $\Pi_{\mathsf{Bootstrap}}$, and let $\delta^*_A = \sum_{j\in A}(\sigma^*_j + \delta'_j)$. That is, δ^*_A should match the sum of the corrupt portion of δ-shares generated in **Initialize**. Now, say P_i inputs bit x to $\Pi_{\mathsf{Bootstrap}}$, then, shares $\{\nu_k\}_{k\notin A}$ are such that $\sum_{k\notin A}\nu_k = \sum_{j\in A}\nu^*_j + x\cdot(\delta^*_A + \delta_H)$. In other words, honest ν_k is consistent with both, δ^*_A (that the adversary imposes via the σ^*_j's) and ν^*_j (that the adversary is suppose to derive from the bits d_j), and these shares are extracted by $\mathcal{S}$ in steps 1 and 3 respectively.

Case dishonest P_i. In this case, $\mathcal{S}$ sends random σ_k to $\mathcal{Z}$ on behalf of honest P_k. This is indistinguishable from what is sent in a real run, as P_k is using a padding given by $\mathcal{F}_{\mathsf{aBit}}$. For what $\Pi_{\mathsf{Bootstrap}}$ outputs to honest parties, we note again that, if $\mathcal{Z}$ gave correct d^*_k to $\mathcal{S}$ using $\mathcal{F}_{\mathsf{AT}}$, the sum of the honest portion of ν-shares is equal to $\sum_{j\in A}(\nu_{i,j} + (x + r_{i,j})\cdot\delta^*_j) + x\cdot\delta_i + \sum_{j\neq i}\mu_{i,j}$, which is extracted by $\mathcal{S}$ in step 1. And if $\mathcal{Z}$ does not send correct d^*_k, namely $d^*_k = x + r_{i,k} + 1$, it would cause honest P_k to compute shifted $\nu_k + \delta_k$, which is exactly what $\mathcal{S}$ tells to $\mathcal{F}_{\mathsf{Bootstrap}}$ to output in step 3. □

C.2 Functionality and Proof of the Online Phase (Theorem 1)

Functionality $\mathcal{F}_{\mathsf{Online}}$

Initialize: On input (*init*) the functionality activates and waits for an input from the environment. Then it does the following: if it receives Abort, it waits for the environment to input a set of corrupted parties, outputs it to the parties, and aborts; otherwise it continues.

Input: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with *varid* a fresh identifier, the functionality stores $(varid, x)$.

Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x + y)$.

Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x \cdot y)$.

Output: On input $(output, varid)$ from all honest parties (if *varid* is present in memory), the functionality retrieves $(varid, y)$ and outputs it to the environment. The functionality waits for an input from the environment. If this input is Deliver then y is output to all players. Otherwise it outputs $\varnothing$ is output to all players.

Figure 12 Secure Function Evaluation

We construct a simulator $\mathcal{S}$ such that an environment $\mathcal{Z}$ corrupting up to $n-1$ parties cannot distinguish whether it is playing with the Π_{Online} attached with $\mathcal{F}_{\mathsf{Prep}}$ and $\mathcal{F}_{\mathsf{Comm}}$, or with the simulator $\mathcal{S}$ and $\mathcal{F}_{\mathsf{Online}}$. We start describing the behaviour of the simulator $\mathcal{S}$:

- The simulation of the **Initialize** procedure is performed running a copy of $\mathcal{F}_{\mathsf{Prep}}$ on query Init. All the data of the corrupted parties are known to the simulator. If $\mathcal{Z}$ inputs Abort to the copy of $\mathcal{F}_{\mathsf{Prep}}$, then the simulator does the same to $\mathcal{F}_{\mathsf{Online}}$ and forward the output of $\mathcal{F}_{\mathsf{Online}}$ to $\mathcal{Z}$: If $\mathcal{F}_{\mathsf{Online}}$ outputs Abort, the simulator waits for input a set of corrupted parties from $\mathcal{Z}$ and forward it to $\mathcal{F}_{\mathsf{Online}}$, and aborts; otherwise it uses the $\mathcal{Z}$'s inputs as preprocessed data.
- In the **Input** stage the simulator does the following. For the honest parties this step is run correctly with dummy inputs; it reads the inputs of corrupted parties specified by $\mathcal{Z}$. Then the simulator runs a copy of Share command of $\mathcal{F}_{\mathsf{Prep}}$ sending back sharings $[x]^i_\alpha$ such that

$i \in A$, where A is the set of corrupted parties. When $\mathcal{Z}$ writes the outputs corresponding to the corrupted parties, the simulator writes these values on the influence port of $\mathcal{F}_{\mathsf{Online}}$ as inputs.
- The procedure **Add, Multiply** are performed according to the protocol and the simulator calls the respective procedure to $\mathcal{F}_{\mathsf{Online}}$.
- In the **Output** step, the functionality $\mathcal{F}_{\mathsf{Online}}$ outputs y to the $\mathcal{S}$. Now the simulator has to provide shares of honest parties such that they are consistent with y. It knows an output value y' computed using the dummy inputs for the honest parties, so it can select a random honest player and modify its share adding $y - y'$ and modify the MAC adding $\alpha(y - y')$, which is possible for the simulator, since it knows α. After that the simulator opens y as in the protocol. If y passes the check, the simulator sends $\mathsf{Deliver}$ to $\mathcal{F}_{\mathsf{Online}}$.

All the steps of the protocol are perfectly simulated: during the initialization the simulator acts as $\mathcal{F}_{\mathsf{Prep}}$; addition does not involve communication, while multiplication implies partial opening: in the protocol, as well as in the simulation, this opening reveals uniform values. Also, MACs have the same distributions in both the protocol and the simulation.

Finally, in the output stage, $\mathcal{Z}$ can see y and the shares from honest parties, which are uniform and compatible with y and its MAC. Moreover it is a correct evaluation of the function on the inputs provided by the parties in the input stage. The same happens in the protocol with overwhelming probability, since the probability that a corrupted party is able to cheat in a $\mathsf{MACCheck}$ call is $2/|\mathbb{F}|$ (see Theorem 4). □

C.3 Proof of the Preprocessing (Theorem 2)

The description of the simulator, denoted by $\mathcal{S}$, is provided in Figure 13. Define $\mathcal{T}_{Real}$ to be the set of messages sent or received from corrupt parties together with the inputs and outputs of the parties, in an execution of Π_{Prep} with $\mathcal{F}_{\mathsf{Bootstrap}}$ and $\mathcal{F}_{\mathsf{Comm}}$. Likewise define $\mathcal{T}_{Ideal}$ for an execution of $\mathcal{F}_{\mathsf{Prep}}$with $\mathcal{S}$. To prove UC security, we see $\mathcal{Z}$ as a distinguisher between the two systems, and our aim is to show that

$$|\Pr[0 \leftarrow \mathcal{Z}(\mathcal{T}_{Real})] - \Pr[0 \leftarrow \mathcal{Z}(\mathcal{T}_{Ideal})] - \frac{1}{2}| \leq \mathsf{negl}(\kappa).$$

For this to hold, it is enough to show that $\mathcal{Z}$ receives as inputs transcripts $\mathcal{T}_{Real}$, $\mathcal{T}_{Ideal}$ that are statistically indistinguishable. We argue as follows.

First note that transcripts generated on calls to **Initialize** and **Share** in both executions, are *perfectly* indistinguishable, as they are nothing but calls to $\mathcal{F}_{\mathsf{Bootstrap}}$ in the real case, with identical behaviour of Share command in $\mathcal{F}_{\mathsf{Prep}}$, (and $\mathcal{S}$ only forwards queries to the $\mathcal{F}_{\mathsf{Prep}}$).

We turn now to GaOT command. Let $\mathsf{OT}_{\mathsf{out}} = \{[\![e]\!], [\![z]\!], [\![x_0]\!], [\![x_1]\!]\}$ be the quadruple that honest parties are hoping to output if no abort occurs. Define the "multiplicative relation" $m = z + x_0 + e \cdot (x_0 + x_1)$, and say that $\mathsf{OT}_{\mathsf{out}}$ is bad if $m = 1$. Thus, bad quadruples are those that implement the multiplication gate incorrectly. Additionally, say that quadruple is noauth if $\mathcal{Z}$ sent to $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ flag $\mathsf{shift\text{-}P_k}$ set to true for at least one honest party P_k, during the execution of **AuthenticateOT**.

Indistinguishability of transcripts. First notice that $\mathcal{T}_{Real}$ and $\mathcal{T}_{Ideal}$ truncated up to the point where the parties output the quadruple are perfectly indistinguishable (steps 12 and 5 respectively): looking at Figure 13, we see that $\mathcal{S}$ sacrifices quadruples exactly as Π_{Prep}. More precisely, step 5 of $\mathcal{S}$ mimics steps 8-12 of Π_{Prep}. Moreover, $\mathcal{S}$ uses quadruples generated in step 3, and honest

The Simulator of Π_{Prep}

The set of corrupt parties is denoted with A.

Initialize: $\mathcal{S}$ forwards to $\mathcal{F}_{\mathsf{Prep}}$ the query (Init) together with $\{\alpha_j\}_{j\in A}$) that $\mathcal{Z}$ does to $\mathcal{F}_{\mathsf{Bootstrap}}$. Then samples random $\alpha \in \mathbb{F}$, and a set of sharings $\{\alpha_k\}_{k\notin A}$ consistent with $\{\alpha_j\}_{j\in A}$ and α, but otherwise random. It stores the complete sharing for later use.

Share: $\mathcal{S}$ forwards to $\mathcal{F}_{\mathsf{Prep}}$ the query (i, Share) of $\mathcal{Z}$ to $\mathcal{F}_{\mathsf{Bootstrap}}$. $\mathcal{S}$ also gets flags $\{\mathsf{shift\text{-}P_k}\}_{k\notin A}$, and MAC shares $\{\mu_j\}_{j\in A}$ from $\mathcal{Z}$. If $i \in A$, $\mathcal{Z}$ specifies input bit x. $\mathcal{S}$ sends shift flags, MAC shares and (possibly) input x to $\mathcal{F}_{\mathsf{Prep}}$.

GaOT:

1. In steps 1 and 3, when $\mathcal{Z}$ thinks is querying $\mathcal{F}_{\mathsf{Bootstrap}}$, on commands Init and Share, respectively, $\mathcal{S}$ discards all the values received from $\mathcal{Z}$.
2. Steps 2, 4, 5 are local, and $\mathcal{S}$ does nothing.
3. Steps 6-7, are repeated four times, one for each symbol $y \in \{e, z, x_0, x_1\}$. In each invocation $\mathcal{S}$ does:
   - During the i-th query to Share command of $\mathcal{F}_{\mathsf{Bootstrap}}$, $\mathcal{S}$ receives from $\mathcal{Z}$ bits $\{y_{i,j}\}_{i\in A}$ and MAC shares $\{\nu_j(y_i) \in \mathbb{F}\}_{j\in A}$, and flags $\{\mathsf{shift\text{-}P_k}^{(i)}\}_{k\notin A}$. It also receives bit y_i, and $\mu(y_i) \in \mathbb{F}$, if $i \in A$.
   - After the n queries are done, $\mathcal{S}$ sets the data of each representation $[y_i]^i_\alpha$ corresponding to honest parties exactly as $\mathcal{F}_{\mathsf{Bootstrap}}$ would do. Thus, if $i \notin A$, $\mathcal{S}$ samples $y_i \in \mathbb{F}_2$, and $\mu(y_i) \in \mathbb{F}$ at random, otherwise uses $\mathcal{Z}$'s choice. It sets $\nu(y_i) = \mu(y_i) + y_i \cdot \alpha$ and prepares sharings $\langle y_i\rangle$ $\langle \nu(y_i)\rangle$ where the honest shares $\nu_k(y_i)$ are consistent with $\mathcal{Z}$'s shares. Finally, $\mathcal{S}$ shifts honest share $\nu_k(y_i) = \nu'_k(y_i)+\alpha_k$ if $\mathsf{shift\text{-}}(P_k)^{(i)}$ is true. The honest data on the joint representation $[\![y]\!]$ is generated as one expects, where $y = \sum_{i\in\mathcal{P}} y_i$.
4. The above steps are repeated at least $\kappa + 1$ times, as in Π_{Prep}.
5. Steps 8-12 are performed as in Π_{Prep}, where $\mathcal{S}$ acts on behalf of honest parties using the dummy quadruples generated in the executions of step 3. It also answers queries from $\mathcal{Z}$ to Comm and Open commands of $\mathcal{F}_{\mathsf{Comm}}$. Openings on behalf of honest parties are set to random seed values.
6. If some iteration in the previous step result in abort, $\mathcal{S}$ inputs Abort to $\mathcal{F}_{\mathsf{Prep}}$. Otherwise, inputs Continue, and for each bit $y \in \{e, z, x_0, x_1\}$ of the checked quadruple, $\mathcal{S}$ discards the shift flags, and gives bit shares $\{y_j\}_{j\in A}$, and MAC shares $\{\mu_j(y)\}_{j\in A}$ derived in step 3, to $\mathcal{F}_{\mathsf{Prep}}$.

Figure 13 The Simulator of Π_{Prep}

parties use quadruples generated in steps 6-7. These quadruples are identically distributed because $\mathcal{S}$ proceeds exactly as $\mathcal{F}_{\mathsf{Bootstrap}}$ does. Also, notice that in Π_{Prep} the output quadruples are those that parties choose to authenticate, and hence $\mathcal{S}$ skips the simulation of **ShareOT** (besides accepting $\mathcal{Z}$'s queries) since no outgoing communication from either $\mathcal{F}_{\mathsf{Bootstrap}}$ or party-to-party is done.

Output indistinguishability. If $\mathcal{Z}$ is honest-but-curious, then a run with Π_{Prep} outputs a quadruple that is neither bad nor noauth. This follows from the correctness of **ShareOT** and **AuthenticateOT** steps. Also, in step 3, $\mathcal{S}$ is able to extract the portion of shares of $\mathsf{OT_{out}}$ corresponding to corrupt parties, and give them to $\mathcal{F}_{\mathsf{Prep}}$. We therefore conclude that the outputs in both worlds are identically distributed. On the other hand, if $\mathcal{Z}$ misbehaves in an arbitrary way, it suffices to show the following to conclude the proof:

$$\mathsf{OT_{out}} \text{ is } \mathsf{bad} \vee \mathsf{noauth} \Rightarrow \Pi_{\mathsf{Prep}} \text{ outputs } \varnothing \text{ with probability } 1 - \mathsf{negl}(\kappa).$$

We argue as follows: the sacrifice step is run by the honest parties. Therein, in the ith iteration, a fresh check quadruple OT_i is taken and honest parties reveal a linear combination on their portion of the shares of $\mathsf{OT_{out}}$ and OT_i, that open to p_i, q_i and c_i. If $\mathcal{Z}$ started with input shares that render an $\mathsf{OT_{out}}$ that is noauth, or chooses to reveal something different, say wlog, the first opening gives wrong p_i^* Then he managed to either pass Π_{MACCheck} on the open values with p_i^* not authenticated, or he managed to authenticate p_i^* and feed it to Π_{MACCheck}. The former happens with probability $\frac{2}{|\mathbb{F}|}$

by Theorem 4 (assuming $\mathsf{PRF}_s^{\mathbb{F},t}(\cdot)$), and the latter is equivalent to have $\mathcal{Z}$ holding the field element $\mu_H + p_i^* \cdot \alpha_H = \sum_{k \notin A}(\mu_k(p_i^*) + p_i^* \cdot \alpha_k)$, and this happens with probability $\frac{1}{|\mathbb{F}|}$, since $\mu_H + p_i^* \cdot \alpha_H$ is only derivable from the private transcripts of honest parties (thus, $\mathcal{Z}$ must guess it). We conclude that, if Π_{MACCheck} passes, then $\mathcal{Z}$ misbehaves in the sacrifice step, or it inputs shares that render an $\mathsf{OT}_{\mathsf{out}}$ that is noauth, with probability bounded by $\frac{2}{|\mathbb{F}|} = 2^{-\kappa+1}$. Now, it is easy to see that if $\mathcal{Z}$ follows the sacrifice step, then we can write $c_i = m \cdot t_i + m'_i$, where m'_i is the multiplicative relation of OT_i. Therefore, if $\mathcal{Z}$ misbehaved in any previous step, yielding $\mathsf{bad}\ \mathsf{OT}_{\mathsf{out}}$, then $c_i = t_i + m'_i$. In this way if the sacrifice step passes, we can write $\mathbf{t} = \mathbf{m}'$, where $\mathbf{t}$ is the challenge vector. This vector is randomly sampled from $\mathbb{F}_2^\kappa$, assuming $\mathsf{PRF}_s^{\mathbb{F}_2,\kappa}(\cdot)$, thus the probability of having $\mathbf{t}$ fixed to $\mathbf{m}'$ is $2^{-\kappa}$.

Summing up, bad or noauth output quadruples will pass both tests with probability at most $2^{-\kappa+1}$. This concludes the proof of the theorem. □

Reducing the Overhead of MPC over a Large Population

A. Choudhury[1], A. Patra[2], and N. P. Smart[3]

[1] IIIT Bangalore, India.
[2] Dept. of Computer Science & Automation, IISc Bangalore, India.
[3] Dept. of Computer Science, Uni. Bristol, United Kingdom.
partho31@gmail.com, arpita@csa.iisc.ernet.in, nigel@cs.bris.ac.uk.

Abstract. We present a secure honest majority MPC protocol, against a static adversary, which aims to reduce the communication cost in the situation where there are a large number of parties and the number of adversarially controlled parties is relatively small. Our goal is to reduce the usage of point-to-point channels among the parties, thus enabling them to run multiple different protocol executions. Our protocol has highly efficient theoretical communication cost when compared with other protocols in the literature; specifically the circuit-dependent communication cost, for circuits of suitably large depth, is $\mathcal{O}(|\mathsf{ckt}|\kappa^7)$, for security parameter κ and circuit size $|\mathsf{ckt}|$. Our protocol finds application in cloud computing scenario, where the fraction of corrupted parties is relatively small. By minimizing the usage of point-to-point channels, our protocol can enable a cloud service provider to run multiple MPC protocols.

1 Introduction

Threshold secure multi-party computation (MPC) is a fundamental problem in secure distributed computing. It allows a set of n mutually distrusting parties with private inputs to "securely" compute any publicly known function of their private inputs, even in the presence of a centralized adversary who can control any t out of the n parties and force them to behave in any arbitrary manner. Now consider a situation, where n is very large, say $n \geq 1000$ and the proportion of corrupted parties (namely the ratio t/n) is relatively small, say 5 percent. In such a scenario, involving all the n parties to perform an MPC calculation is wasteful, as typical (secret-sharing based) MPC protocols require all parties to simultaneously transmit data to all other parties. However, restricting to a small subset of parties may lead to security problems. In this paper we consider the above scenario and show how one can obtain a communication efficient, robust MPC protocol which is actively secure against a computationally bounded static adversary. In particular we present a protocol in which the main computation is performed by a "smallish" subset of the parties, with the whole set of parties used occasionally so as to "checkpoint" the computation. By not utilizing the entire set of parties all the time enables them to run many MPC calculations at once. The main result we obtain in the paper is as follows:

> **Main Result (Informal):** Let $\epsilon = \frac{t}{n}$ with $0 \leq \epsilon < 1/2$ and let the t corrupted parties be under the control of a computationally bounded static adversary. Then for a security parameter κ (for example $\kappa = 80$ or $\kappa = 128$), there exists an MPC protocol with the following circuit-dependent communication complexity[4] to evaluate an arithmetic circuit ckt: **(a).** $\mathcal{O}(|\mathsf{ckt}| \cdot \kappa^7)$ for ckt with depth $\omega(t)$. **(b).** $\mathcal{O}(|\mathsf{ckt}| \cdot \kappa^4)$ for ckt with $d = \omega(t)$ and $w = \omega(\kappa^3)$ (i.e. $|\mathsf{ckt}| = \omega(\kappa^3 t)$).

Protocol Overview: We make use of two secret-sharing schemes. A secret-sharing scheme $[\cdot]$ which is an actively-secure variant of the Shamir secret-sharing scheme [28] with threshold t. This first secret-sharing

[4] The communication complexity of an MPC protocol has two parts: a circuit-dependent part, dependent on the circuit size and a circuit-independent part. The focus is on the circuit-dependent communication, based on the assumption that the circuit is large enough so that the terms independent of the circuit-size can be ignored; see for example [11, 4, 12, 5].

scheme is used to share values amongst *all* of the n parties. The second secret-sharing scheme $\langle\cdot\rangle$ is an actively-secure variant of an additive secret-sharing scheme, amongst a well-defined subset $\mathcal{C}$ of the parties.

Assuming the inputs to the protocol are $[\cdot]$ shared amongst the parties at the start of the protocol, we proceed as follows. We first divide ckt into L levels, where each level consists of a sub-circuit. The computation now proceeds in L phases; we describe phase i. At the start of phase i we have that *all* n parties hold $[\cdot]$ sharings of the inputs to level i. The n parties then select (at random) a committee $\mathcal{C}$ of size $\mathfrak{c}$. If $\mathfrak{c}$ is such that $\epsilon^{\mathfrak{c}} < 2^{-\kappa}$ then statistically the committee $\mathcal{C}$ will contain at least one honest party, as the inequality implies that the probability that the committee contains no honest party is negligibly small. The n parties then engage in a "conversion" protocol so that the input values to level i are now $\langle\cdot\rangle$ shared amongst the committee. The committee $\mathcal{C}$ then engages in an actively-secure dishonest majority[5] MPC protocol to evaluate the sub-circuit at level i. If no abort occurs during the evaluation of the ith sub-circuit then the parties engage in another "conversion" protocol so that the output values of the sub-circuit are converted from a $\langle\cdot\rangle$ sharing amongst members in $\mathcal{C}$ to a $[\cdot]$ sharing amongst all n parties. This step amounts to check-pointing data. This ensures that the inputs to all the subsequent sub-circuits are saved in the form of $[\cdot]$ sharing which guarantees recoverability as long as $0 \leq \epsilon < \frac{1}{2}$. So the check-pointing prevents from re-evaluating the entire circuit from scratch after every abort of the dishonest-majority MPC protocol.

If however an abort occurs while evaluating the ith sub-circuit then we determine a pair of parties from the committee $\mathcal{C}$, one of whom is guaranteed to be corrupted and eliminate the pair from the set of active parties, and re-evaluate the sub-circuit again. In fact, cheating can also occur in the $\langle\cdot\rangle \leftrightarrow [\cdot]$ conversions and we need to deal with these as well. Thus if errors are detected we need to repeat the evaluation of the sub-circuit at level i. Since there are at most t bad parties, the total amount of backtracking (i.e. evaluating a sub-circuit already computed) that needs to be done is bounded by t. For large n and small t this provides an asymptotically efficient protocol.

The main technical difficulty is in providing actively-secure conversions between the two secret-sharing schemes, and providing a suitable party-elimination strategy for the dishonest majority MPC protocol. The party-elimination strategy we employ follows from standard techniques, as long as we can identify the pair of parties. This requirement, of a dishonest-majority MPC protocol which enables identification of cheaters, without sacrificing privacy, leads us to the utilization of the protocol in [12]. This results in us needing to use double-trapdoor homomorphic commitments as a basic building block. To ensure greater asymptotic efficiency we apply two techniques: **(a).** the check-pointing is done among a set of parties that assures honest majority with overwhelming probability **(b).** the packing technique from [20] to our Shamir based secret sharing.

To obtain an efficient protocol one needs to select L; if L is too small then the sub-circuits are large and so the cost of returning to a prior checkpoint will also be large. If however L is too large then we will need to checkpoint a lot, and hence involve all n parties in the computation at a lot of stages (and thus requiring all n parties to be communicating/computing). The optimal value of L for our protocol turns out to be t.

Related Work: The circuit-dependent communication complexity of the traditional MPC protocols in the honest-majority setting is $\mathcal{O}(|\mathsf{ckt}| \cdot \mathsf{Poly}(n, \kappa))$; this informally stems from the fact in these protocols we require all the n parties to communicate with each other for evaluating each gate of the circuit. Assuming $0 \leq \epsilon < 1/2$, [11] presents a computationally secure MPC protocol with communication complexity $\mathcal{O}(|\mathsf{ckt}| \cdot \mathsf{Poly}(\kappa, \log n, \log |\mathsf{ckt}|))$. The efficiency comes from the ability to pack and share several values simultaneously which in turn allow parallel evaluation of "several" gates simultaneously in a single round

[5] In the dishonest-majority setting, the adversary may corrupt all but one parties. An MPC protocol in this setting aborts if a corrupted party misbehaves.

of communication. However, the protocol still requires communications between all the parties during each round of communication. Our protocol reduces the need for the parties to be communicating with all others at all stages in the protocol; moreover, asymptotically for large n it provides a better communication complexity over [11] (as there is no dependence on n), for circuits of suitably large depth as stated earlier. However, the protocol of [11] is secure against a more powerful adaptive adversary.

In the literature, another line of investigation has been carried out in [6, 10, 14, 15] to beat the $\mathcal{O}(|\mathsf{ckt}| \cdot \mathsf{Poly}(n, \kappa))$ communication complexity bound of traditional MPC protocols, against a static adversary. The main idea behind all these works is similar to ours, which is to involve "small committees" of parties for evaluating each gate of the circuit, rather than involving all the n parties. The communication complexity of these protocols[6] is of the order $\mathcal{O}(|\mathsf{ckt}| \cdot \mathsf{Poly}(\log n, \kappa))$. Technically our protocol is different from these protocols in the following ways: **(a).** The committees in [6, 10, 14, 15] are of size $\mathsf{Poly}(\log n)$, which ensures that with high probability the selected committees have honest majority. As a result, these protocols run any existing honest-majority MPC protocol among these small committees of $\mathsf{Poly}(\log n)$ size, which prevents the need to check-point the computation (as there will be no aborts). On the other hand, we only require committees with at least one honest party and our committee size is independent of n, thus providing better communication complexity. Indeed, asymptotically for large n, our protocol provides a better communication complexity over [6, 10, 14, 15] (as there is no dependence on n), for circuits of suitably large depth. **(b).** Our protocol provides a better fault-tolerance. Specifically, [14, 10, 6] requires $\epsilon < 1/3$ and [15] requires $\epsilon < 1/8$; on the other hand we require $\epsilon < 1/2$.

We stress that the committee selection protocol in [6, 10, 14, 15] is unconditionally secure and in the full-information model, where the corrupted parties can see all the messages communicated between the honest parties. On the other hand our implementation of the committee selection protocol is computationally secure. The committee election protocol in [6, 10, 14, 15] is inherited from [17]. The committee selection protocol in these protocols are rather involved and not based on simply randomly selecting a subset of parties, possibly due to the challenges posed in the full information model with unconditional security; this causes their committee size to be logarithmic in n. However, if one is willing to relax at least one of the above two features (i.e. full information model and unconditional security), then it may be possible to select committees with honest majority in a simple way by randomly selecting committees, where the committee size may be independent of n. However investigating the same is out of the scope of this paper.

Finally we note that the idea of using small committees has been used earlier in the literature for various distributed computing tasks, such as the leader election [23, 26], Byzantine agreement [24, 25] and distributed key-generation [9].

On the Choice of ϵ: We select committees of size $\mathfrak{c}$ satisfying $\epsilon^{\mathfrak{c}} < 2^{-\kappa}$. This implies that the selected committee has at least one honest participant with overwhelming probability. We note that it is possible to randomly select committees of "larger" size so that with overwhelming probability the selected committee will have honest majority. We label the protocol which samples a committee with honest majority and then runs an computationally secure honest majority MPC protocol (where we need not have to worry about aborts) as the "naive protocol". The naive protocol will have communication complexity $\mathcal{O}(|\mathsf{ckt}| \cdot \mathsf{Poly}(\kappa))$.

For "very small" values of ϵ, the committee size for the naive protocol is comparable to the committee size in our protocol. We demonstrate this with an example, with $n = 1000$ and security level $\kappa = 80$: The committee size we require to ensure both a single honest party in the committee and a committee with honest majority, with overwhelming probability of $(1 - 2^{-80})$ for various choices of ϵ, is given in the following table:

[6] Note, the protocol of [6] involves FHE to further achieve a communication complexity of $\mathcal{O}(\mathsf{Poly}(\log n))$.

ϵ	$\mathfrak{c}$ to obtain at least one honest party	$\mathfrak{c}$ to obtain honest majority
$1/3$	48	448
$1/4$	39	250
$1/10$	23	84
$1/100$	11	20

From the table it is clear that when ϵ is closer to $1/2$, the difference in the committee size to obtain at least one honest party and to obtain honest majority is large. As a result, selecting committees with honest majority can be prohibitively expensive, thus our selection of small committees with dishonest majority provides significant improvements.

To see intuitively why our protocol selects smaller committees, consider the case when the security parameter κ tends to infinity: Our protocol will require a committee of size roughly $\epsilon \cdot n + 1$, whereas the naive protocol will require a committee of size roughly $2 \cdot \epsilon \cdot n + 1$. Thus the naive method will use a committee size of roughly twice that of our method. Hence, if small committees are what is required then our method improves on the naive method.

For fixed ϵ and increasing n, we can apply the binomial approximation to the hypergeometric distribution, and see that our protocol will require a committee of size $c \approx \kappa / \log_2(\frac{1}{\epsilon})$. To estimate the committee size for the naive protocol we use the cumulative distribution function for the binomial distribution, $F(b; c, \epsilon)$, which gives the probability that we select at least b corrupt parties in a committee of size c given the probability of a corrupt party being fixed at ϵ. To obtain an honest majority with probability less than $2^{-\kappa}$ we require $F(c/2; c, \epsilon) \approx 2^{-\kappa}$. By estimating $F(c/2; c, \epsilon)$ via Hoeffding's inequality we obtain

$$\exp\left(-2 \cdot \frac{(c \cdot \epsilon - c/2)^2}{c}\right) \approx 2^{-\kappa},$$

which implies

$$\kappa \approx \left(\frac{c \cdot (2 \cdot \epsilon - 1)^2}{2}\right) / \log_e 2.$$

Solving for c gives us

$$c \approx \frac{2 \cdot \kappa \cdot \log_e 2}{(2 \cdot \epsilon - 1)^2}.$$

Thus for fixed ϵ and large n the number of parties in a committee is $O(\kappa)$ for both our protocol, and the naive protocol. Thus the communication complexity of our protocol and the naive protocol is asymptotically the same. But, since the committees in our protocol are always smaller than those in the naive protocol, we will obtain an advantage when the ratio of the different committee size is large, i.e. when ϵ is larger.

The the ratio between the committee size in the naive protocol and that of our protocol (assuming we are in a range when Hoeffding's inequality provides a good approximation) is roughly

$$\frac{-2 \cdot \log_e 2 \cdot \log_2 \epsilon}{(2 \cdot \epsilon - 1)^2}$$

So for large n the ratio between the committee sizes of the two protocols depends on ϵ alone (and is independent of κ). By way of example this ratio is approximately equal to 159 when $\epsilon = 0.45$, 19 when $\epsilon = 1/3$, 7 when $\epsilon = 1/10$ and 9.6 when $\epsilon = 1/100$; although the approximation via Hoeffding's inequality only really applies for ϵ close to $1/2$.

This implies that for values of ϵ close to $1/2$ our protocol will be an improvement on the naive protocol. However, the naive method does not have the extra cost of check-pointing which our method does; thus at some point the naive protocol will be more efficient. Thus our protocol is perhaps more interesting, when ϵ is not too small, say in the range of $[1/100, 1/2]$.

Possible Application of Our Protocol for Cloud-Computing. Consider the situation of an organization performing a multi-party computation on a cloud infrastructure, which involves a large number of machines, with the number of corrupted parties possibly high, but not exceeding one half of the parties, (which is exactly the situation considered in our MPC protocol). Using our MPC protocol, the whole computation can be then carried out by a small subset of machines, with the whole cloud infrastructure being used only for check-pointing the computation. By not utilizing the whole cloud infrastructure all the time, we enable the cloud provider to serve multiple MPC requests.

Our protocol is not adaptively secure. In fact, vulnerability to adaptive adversary is inherent to most of the committee-based protocols for several distributed computing tasks such as Leader Election [23, 26], Byzantine Agreement [25, 24], Distributed Key-generation [9] and MPC in [14, 10]. Furthermore, We feel that adaptive security is not required in the cloud scenario. Any external attacker to the cloud data centre will have a problem determining which computers are being used in the committee, and an even greater problem in compromising them adaptively. The main threat model in such a situation is via co-tenants (other users processes) to be resident on the same physical machine. Since the precise machine upon which a cloud tenant sits is (essentially) randomly assigned, it is hard for a co-tenant adversary to mount a cross-Virtual Machine attack on a specific machine unless they are randomly assigned this machine by the cloud. Note, that co-tenants have more adversarial power than a completely external attacker. A more correct security model would be to have a form of adaptive security in which attackers pro-actively move from one machine to another, but in a random fashion. We leave analysing this complex situation to a future work.

2 Model, Notation and Preliminaries

We denote by $\mathcal{P} = \{P_1, \ldots, P_n\}$ the set of n parties who are connected by pair-wise private and authentic channels. We assume that there exists a PPT static adversary $\mathcal{A}$, who can maliciously corrupt any t parties from $\mathcal{P}$ at the beginning of the execution of a protocol, where $t = n \cdot \epsilon$ and $0 \leq \epsilon < \frac{1}{2}$. There exists a publicly known randomized function $f : \mathbb{F}_p^n \rightarrow \mathbb{F}_p$, expressed as a publicly known arithmetic circuit ckt over the field $\mathbb{F}_p$ of prime order p (including random gates to enable the evaluation of randomized functions), with party P_i having a private input $x^{(i)} \in \mathbb{F}_p$ for the computation. We let d and w to denote the depth and (average) width of ckt respectively. The finite field $\mathbb{F}_p$ is assumed to be such that p is a prime, with $p > \max\{n, 2^\kappa\}$, where κ is the *computational security parameter*. Apart from κ, we also have an additional *statistical security parameter* s and the security offered by s (which is generally much smaller than κ) does not depend on the computational power of the adversary.

The security of our protocol(s) will be proved in the universal composability (UC) model. The UC framework allows for defining the security properties of cryptographic tasks so that security is maintained under general composition with an unbounded number of instances of arbitrary protocols running concurrently. In the framework, the security requirements of a given task are captured by specifying an ideal functionality run by a "trusted party" that obtains the inputs of the parties and provides them with the desired outputs. Informally, a protocol securely carries out a given task if running the protocol in the presence of a real-world adversary amounts to "emulating" the desired functionality. For more details, see Appendix A.

We do not assume a physical broadcast channel. Although our protocol uses an ideal broadcast functionality $\mathcal{F}_{\text{BC}}$ (Fig. 3), that allows a sender $\mathsf{Sen} \in \mathcal{P}$ to reliably broadcast a message to a group of parties $\mathcal{X} \subseteq \mathcal{P}$, the functionality can be instantiated using point-to-point channels; see Appendix B.2 for details.

The communication complexity of our protocols has two parts: the communication done over the point-to-point channels and the broadcast communication. The later is captured by $\mathcal{BC}\big(\ell, |\mathcal{X}|\big)$ to denote that in total, $\mathcal{O}(\ell)$ bits is broadcasted in the associated protocol to a set of parties of size $|\mathcal{X}|$. For details about the instantiation of $\mathcal{F}_{\text{BC}}$, see Appendix B.

Two different types of secret-sharing are employed in our protocols. The secret-sharings are inherently defined to include "verification information" of the individual shares in the form of publicly known commitments. We use a variant of the Pedersen homomorphic commitment scheme [27]. In our protocol, we require UC-secure commitments to ensure that a committer must know its committed value and just cannot manipulate a commitment produced by other committers to violate what we call "input independence". It has been shown in [8] that a UC secure commitment scheme is impossible to achieve without setup assumptions. The standard method to implement UC-secure commitments is in the Common Reference String (CRS) model where it is assumed that the parties are provided with a CRS that is set up by a "trusted third party" (TTP). We follow [12], where the authors show how to build a multiparty UC-secure homomorphic commitment scheme (where multiple parties can act as committer) based on any double-trapdoor homomorphic commitment scheme.

Definition 1 (Double-trapdoor Homomorphic Commitment for $\mathbb{F}_p$ [12]). *It is a collection of five PPT algorithms* $(\mathsf{Gen}, \mathsf{Comm}, \mathsf{Open}, \mathsf{Equivocate}, \mathsf{TDExtract}, \odot)$*:*

- $\mathsf{Gen}(1^\kappa) \rightarrow (\mathsf{ck}, \tau_0, \tau_1)$*: the generation algorithm outputs a commitment key* ck*, along with trapdoors* τ_0 *and* τ_1*.*
- $\mathsf{Comm}_{\mathsf{ck}}(x; r_0, r_1) \rightarrow \mathbf{C}_{x,r_0,r_1}$*: the commitment algorithm takes a message* $x \in \mathbb{F}_p$ *and randomness* r_0, r_1 *from the commitment randomness space* $\mathcal{R}$ [7] *and outputs a commitment* $\mathbf{C}_{x;r_0,r_1}$ *of* x *under the randomness* r_0, r_1*.*
- $\mathsf{Open}_{\mathsf{ck}}(\mathbf{C}, (x; r_0, r_1)) \rightarrow \{0,1\}$*: the opening algorithm takes a commitment* $\mathbf{C}$*, along with a message/randomness triplet* (x, r_0, r_1) *and outputs* 1 *if* $\mathbf{C} = \mathsf{Comm}_{\mathsf{ck}}(x; r_0, r_1)$*, else* 0*.*
- $\mathsf{Equivocate}(\mathbf{C}_{x,r_0,r_1}, x, r_0, r_1, \overline{x}, \tau_i) \rightarrow (\overline{r}_0, \overline{r}_1) \in \mathcal{R}$*: using one of the trapdoors* τ_i *with* $i \in \{0,1\}$*, the equivocation algorithm can open a commitment* $\mathbf{C}_{x,r_0,r_1}$ *with any message* $\overline{x} \neq x$ *with randomness* $\overline{r}_0$ *and* $\overline{r}_1$ *where* $r_{1-i} = \overline{r}_{1-i}$*.*
- $\mathsf{TDExtract}(\mathbf{C}, x, r_0, r_1, \overline{x}, \overline{r}_0, \overline{r}_1, \tau_i) \rightarrow \tau_{1-i}$*: using one of the trapdoors* τ_i *with* $i \in \{0,1\}$ *and two different sets of message/randomness triplet for the same commitment, namely* x, r_0, r_1 *and* $\overline{x}, \overline{r}_0, \overline{r}_1$*, the trapdoor extraction algorithm can find the other trapdoor* τ_{1-i} *if* $r_{1-i} \neq \overline{r}_{1-i}$*.*
  The commitments are homomorphic meaning that $\mathsf{Comm}(x; r_0, r_1) \odot \mathsf{Comm}(y; s_0, s_1) = \mathsf{Comm}(x + y; r_0 + s_0, r_1 + s_1)$ *and* $\mathsf{Comm}(x; r_0, r_1)^c = \mathsf{Comm}(c \cdot x; c \cdot r_0, c \cdot r_1)$ *for any publicly known constant* c*.*

We require the following properties to be satisfied:

- **Trapdoor Security***: There exists no PPT algorithm* A *such that* $A(1^\kappa, \mathsf{ck}, \tau_i) \rightarrow \tau_{1-i}$*, for* $i \in \{0,1\}$*.*
- **Computational Binding***: There exists no PPT algorithm* A *with* $A(1^\kappa, \mathsf{ck}) \rightarrow (x, r_0, r_1, \overline{x}, \overline{r}_0, \overline{r}_1)$ *and* $(x, r_0, r_1) \neq (\overline{x}, \overline{r}_0, \overline{r}_1)$*, but* $\mathsf{Comm}_{\mathsf{ck}}(x; r_0, r_1) = \mathsf{Comm}_{\mathsf{ck}}(\overline{x}; \overline{r}_0, \overline{r}_1)$*.*
- **Statistical Hiding***: $\forall x, \overline{x} \in \mathbb{F}_p$ and $r_0, r_1 \in \mathcal{R}$, let $(\overline{r}_0, \overline{r}_1) = \mathsf{Equivocate}(\mathbf{C}_{x,r_0,r_1}, x, r_0, r_1, \overline{x}, \tau_i)$, with $i \in \{0,1\}$. Then $\mathsf{Comm}_{\mathsf{ck}}(x; r_0, r_1) = \mathsf{Comm}_{\mathsf{ck}}(\overline{x}; \overline{r}_0, \overline{r}_1) = \mathbf{C}_{x,r_0,r_1}$; moreover the distribution of (r_0, r_1) and $(\overline{r}_0, \overline{r}_1)$ are statistically close.*

We will use the following instantiation of a double-trapdoor homomorphic commitment scheme which is a variant of the standard Pedersen commitment scheme over a group $\mathbb{G}$ in which discrete logarithms are hard [12]. The message space is $\mathbb{F}_p$ and the randomness space is $\mathcal{R} = \mathbb{F}_p^2$.

- $\mathsf{Gen}(1^\kappa) \rightarrow ((\mathbb{G}, p, g, h_0, h_1), \tau_0, \tau_1)$, where $\mathsf{ck} = (\mathbb{G}, p, g, h_0, h_1)$ such that g, h_0, h_1 are generators of the group $\mathbb{G}$ of prime order p and $g^{\tau_i} = h_i$ for $i \in \{0,1\}$.

[7] For the ease of presentation, we assume $\mathcal{R}$ to be an additive group.

- $\mathsf{Comm}_{\mathsf{ck}}(x; r_0, r_1) \to g^x h_0^{r_0} h_1^{r_1} = \mathbf{C}_{x,r_0,r_1}$, with $x, r_0, r_1 \in \mathbb{F}_p$.
- $\mathsf{Open}_{\mathsf{ck}}(\mathbf{C}, (x, r_0, r_1)) \to 1$, if $\mathbf{C} = g^x h_0^{r_0} h_1^{r_1}$, else $\mathsf{Open}_{\mathsf{ck}}(\mathbf{C}, (x, r_0, r_1)) \to 0$.
- $\mathsf{Equivocate}(\mathbf{C}_{x,r_0,r_1}, x, r_0, r_1, \overline{x}, \tau_i) \to (\overline{r}_0, \overline{r}_1)$ where $\overline{r}_{1-i} = r_{1-i}$ and $\overline{r}_i = \tau_i^{-1}(x - \overline{x}) + r_i$.
- $\mathsf{TDExtract}(\mathbf{C}, x, r_0, r_1, \overline{x}, \overline{r}_0, \overline{r}_1, \tau_i) \to \tau_{1-i}$, where if $\overline{r}_{1-i} \neq r_{1-i}$, then

$$\tau_{1-i} = \frac{\overline{x} - x + \tau_i(\overline{r}_i - r_i)}{r_{1-i} - \overline{r}_{1-i}}.$$

- The homomorphic operation $\odot$ is just the group operation i.e.

$$\begin{aligned}\mathsf{Comm}(x; r_0, r_1) \odot \mathsf{Comm}(\overline{x}; \overline{r}_0, \overline{r}_1) &= g^x h_0^{r_0} h_1^{r_1} \cdot g^{\overline{x}} h_0^{\overline{r}_0} h_1^{\overline{r}_1} \\ &= g^{x+\overline{x}} \cdot h_0^{r_0+\overline{r}_0} \cdot h_1^{r_1+\overline{r}_1} \\ &= \mathsf{Comm}(x + \overline{x}; r_0 + \overline{r}_0, r_1 + \overline{r}_1).\end{aligned}$$

We can now define the various types of secret-shared data used in our protocols. Let $\alpha_1, \ldots, \alpha_n \in \mathbb{F}_p$ be n publicly known non-zero, distinct values, where α_i is associated with P_i as the *evaluation point*. The $[\cdot]$ sharing is the standard Shamir-sharing [28], where the secret value will be shared among the set of parties $\mathcal{P}$ with threshold t. Additionally, a commitment of each individual share will be available publicly, with the corresponding share-holder possessing the randomness of the commitment.

Definition 2 (The $[\cdot]$ Sharing). *Let $s \in \mathbb{F}_p$; then s is said to be $[\cdot]$-shared among $\mathcal{P}$ if there exist polynomials, say $f(\cdot), g(\cdot)$ and $h(\cdot)$, of degree at most t, with $f(0) = s$ and every (honest) party $P_i \in \mathcal{P}$ holds a share $f_i = f(\alpha_i)$ of s, along with opening information $g_i = g(\alpha_i)$ and $h_i = h(\alpha_i)$ for the commitment $\mathbf{C}_{f_i,g_i,h_i} = \mathsf{Comm}_{\mathsf{ck}}(f_i; g_i, h_i)$. The information available to party $P_i \in \mathcal{P}$ as part of the $[\cdot]$-sharing of s is denoted by $[s]_i = (f_i, g_i, h_i, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}})$. All parties will also have the access to* ck*. Moreover, the collection of $[s]_i$'s, corresponding to $P_i \in \mathcal{P}$ is denoted by $[s]$.*

The second type of secret-sharing (which is a variation of additive sharing), is used to perform computation via a dishonest majority MPC protocol amongst our committees.

Definition 3 (The $\langle\cdot\rangle$ Sharing). *A value $s \in \mathbb{F}_p$ is said to be $\langle\cdot\rangle$-shared among a set of parties $\mathcal{X} \subseteq \mathcal{P}$, if every (honest) party $P_i \in \mathcal{X}$ holds a share s_i of s along with the opening information u_i, v_i for the commitment $\mathbf{C}_{s_i,u_i,u_i} = \mathsf{Comm}_{\mathsf{ck}}(s_i; u_i, v_i)$, such that $\sum_{P_i \in \mathcal{X}} s_i = s$. The information available to party $P_i \in \mathcal{X}$ as part of the $\langle\cdot\rangle$-sharing of s is denoted by $\langle s\rangle_i = (s_i, u_i, v_i, \{\mathbf{C}_{s_j,u_j,v_j}\}_{P_j \in \mathcal{X}})$. All parties will also have access to* ck*. The collection of $\langle s\rangle_i$'s corresponding to $P_i \in \mathcal{X}$ is denoted by $\langle s\rangle_{\mathcal{X}}$.*

It is easy to see that both types of secret-sharing are linear. For example, for the $\langle\cdot\rangle$ sharing, given $\langle s^{(1)}\rangle_{\mathcal{X}}, \ldots, \langle s^{(\ell)}\rangle_{\mathcal{X}}$ and publicly known constants $c_1, \ldots, c_\ell$, the parties in $\mathcal{X}$ can locally compute their information corresponding to $\langle c_1 \cdot s^{(1)} + \ldots + c_\ell \cdot s^{(\ell)}\rangle_{\mathcal{X}}$. This follows from the homomorphic property of the underlying commitment scheme and the linearity of the secret-sharing scheme. This means that the parties in $\mathcal{X}$ can locally compute $\langle c_1 \cdot s^{(1)} + \ldots + c_\ell \cdot s^{(\ell)}\rangle_{\mathcal{X}}$ from $\langle s^{(1)}\rangle_{\mathcal{X}}, \ldots, \langle s^{\ell}\rangle_{\mathcal{X}}$, since each party P_i in $\mathcal{X}$ can locally compute $\langle c_1 \cdot s^{(1)} + \ldots + c_\ell \cdot s^{(\ell)}\rangle_i$ from $\langle s^{(1)}\rangle_i, \ldots, \langle s^{\ell}\rangle_i$.

3 Main Protocol

We now present an MPC protocol implementing the standard honest-majority (meaning $\epsilon < 1/2$) MPC functionality $\mathcal{F}_f$ presented in Figure 1 which computes the function f.

We now present the underlying idea of our protocol (outlined earlier in the introduction). The protocol is set in a variant of the *player-elimination* framework from [4]. During the computation either pairs of parties,

Functionality $\mathcal{F}_f$

$\mathcal{F}_f$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$ and is parametrized by an n-input function $f : \mathbb{F}_p^n \rightarrow \mathbb{F}_p$.

- Upon receiving $(\mathsf{sid}, i, x^{(i)})$ from every $P_i \in \mathcal{P}$ where $x^{(i)} \in \mathbb{F}_p$, the functionality computes $y = f(x^{(1)}, \ldots, x^{(n)})$, sends (sid, y) to all the parties and the adversary $\mathcal{S}$ and halts.

Fig. 1. The Ideal Functionality for Computing a Given Function f

each containing at least one actively corrupted party, or singletons of corrupted parties, are identified due to some adversarial behavior of the corrupted parties. These pairs, or singletons, are then eliminated from the set of eligible parties. To understand how we deal with the active corruptions, we need to define a dynamic set $\mathcal{L} \subseteq \mathcal{P}$ of size $\mathfrak{n}$, which will define the current set of eligible parties in our protocol, and a threshold $\mathfrak{t}$ which defines the maximum number of corrupted parties in $\mathcal{L}$. Initially $\mathcal{L}$ is set to be equal to $\mathcal{P}$ (hence $\mathfrak{n} = n$) and $\mathfrak{t}$ is set to t. We then divide the circuit ckt (representing f) to be evaluated into L levels, where each level consists of a sub-circuit of depth d/L; without loss of generality, we assume d to be a multiple of L. We denote the ith sub-circuit as ckt_i. At the beginning of the protocol, all the parties in $\mathcal{P}$ verifiably $[\cdot]$-share their inputs for the circuit ckt.

For evaluating a sub-circuit ckt_l, instead of involving all the parties in $\mathcal{L}$, we rather involve a small and random committee $\mathcal{C} \subset \mathcal{L}$ of parties of size $\mathfrak{c}$, where $\mathfrak{c}$ is the minimum value satisfying the constraint that $\epsilon^{\mathfrak{c}} \leq 2^{-\kappa}$; recall $\epsilon = t/n$. During the course of evaluating the sub-circuit, if any inconsistency is reported, then the (honest) parties in $\mathcal{P}$ will identify either a single corrupted party or a pair of parties from $\mathcal{L}$ where the pair contains at least one corrupted party. The identified party(ies) is(are) eliminated from $\mathcal{L}$ and the value of $\mathfrak{t}$ is decremented by one, followed by re-evaluation of ckt_l by choosing a new committee from the *updated* set $\mathcal{L}$. This is reminiscent of the player-elimination framework from [4], however the way we apply the player-elimination framework is different from the standard one. Specifically, in the player-elimination framework, the *entire* set of eligible parties $\mathcal{L}$ is involved in the computation and the player elimination is then performed over the entire $\mathcal{L}$, thus requiring huge communication. On the contrary, in our context, only a small set of parties $\mathcal{C}$ is involved in the computation, thus significantly reducing the communication complexity. It is easy to see that after a sequence of t failed sub-circuit evaluations, $\mathcal{L}$ will be left with only honest parties and so each sub-circuit will be evaluated successfully from then onwards.

Note that the way we eliminate the parties, the fraction of corrupted parties in $\mathcal{L}$ after any un-successful attempt for sub-circuit evaluation, is upper bounded by the fraction of corrupted parties in $\mathcal{L}$ prior to the evaluation of the sub-circuit. Specifically, let $\epsilon_{\mathsf{old}} = \mathfrak{t}/\mathfrak{n}$ be the fraction of corrupted parties in $\mathcal{L}$ prior to the evaluation of a sub-circuit ckt_l and let the evaluation fail, with either a single party or a pair of parties being eliminated from $\mathcal{L}$. Moreover, let ϵ_{new} be the fraction of corrupted parties in $\mathcal{L}$ after the elimination. Then for single elimination, we have $\epsilon_{\mathsf{new}} = \frac{\mathfrak{t}-1}{\mathfrak{n}-1}$ and so $\epsilon_{\mathsf{new}} \leq \epsilon_{\mathsf{old}}$ if and only if $\mathfrak{n} \geq \mathfrak{t}$, which will always hold. On the other hand, for double elimination, we have $\epsilon_{\mathsf{new}} = \frac{\mathfrak{t}-1}{\mathfrak{n}-2}$ and so $\epsilon_{\mathsf{new}} \leq \epsilon_{\mathsf{old}}$ if and only if $\mathfrak{n} \geq 2\mathfrak{t}$, which will always hold.

Since a committee $\mathcal{C}$ (for evaluating a sub-circuit) is selected randomly, except with probability at most $\epsilon^{\mathfrak{c}} < 2^{-\kappa}$, the selected committee contains at least one honest party and so the sub-circuit evaluation among $\mathcal{C}$ needs to be performed via a dishonest majority MPC protocol. We choose the MPC protocol of [12], since it can be modified to identify pairs of parties consisting of at least one corrupted party in the case of the failed evaluation, without violating the privacy of the honest parties. To use the protocol of [12] for sub-circuit evaluation, we need the corresponding sub-circuit inputs (available to the parties in $\mathcal{P}$ in $[\cdot]$-shared form) to be converted and available in $\langle\cdot\rangle$-shared form to the parties in $\mathcal{C}$ and so the parties in $\mathcal{P}$ do the same. After every successful evaluation of a sub-circuit, via the dishonest majority MPC protocol, the outputs of that sub-circuit (available in $\langle\cdot\rangle$-shared form to the parties in a committee) are converted and saved in the

form of $[\cdot]$-sharing among all the parties in $\mathcal{P}$. As the set $\mathcal{P}$ has a honest majority, $[\cdot]$-sharing ensures robust reconstruction implying that the shared values are recoverable. Since the inputs to a sub-circuit come either from the outputs of previous sub-circuit evaluations or the original inputs, both of which are $[\cdot]$-shared, a failed attempt for a sub-circuit evaluation does not require a re-evaluation of the entire circuit from scratch but requires a re-evaluation of that sub-circuit only.

3.1 Supporting Functionalities

We now present a number of ideal functionalities defining sub-components of our main protocol; see Appendix B for the UC-secure instantiations of these functionalities.

Basic Functionalities: The functionality $\mathcal{F}_{\text{CRS}}$ for generating the common reference string (CRS) for our main MPC protocol is given in Figure 2. The functionality outputs the commitment key of a double-trapdoor homomorphic commitment scheme, along with the encryption key of an IND-CCA secure encryption scheme (to be used later for UC-secure generation of completely random $\langle\cdot\rangle$-shared values as in [12]), see Appendix D. The functionality $\mathcal{F}_{\text{BC}}$ for group broadcast is given in Figure 3. This functionality broadcasts the message sent by a sender $\mathsf{Sen} \in \mathcal{P}$ to all the parties in a sender specified set of parties $\mathcal{X} \subseteq \mathcal{P}$; in our context, the set $\mathcal{X}$ will always contain at least one honest party. The functionality $\mathcal{F}_{\text{COMMITTEE}}$ for a random committee selection is given in Figure 4. This functionality is parameterized by a value $\mathfrak{c}$, it selects a set $\mathcal{X}$ of $\mathfrak{c}$ parties at random from a specified set $\mathcal{Y}$ and outputs the selected set $\mathcal{X}$ to the parties in $\mathcal{P}$.

Functionality $\mathcal{F}_{\text{CRS}}$

$\mathcal{F}_{\text{CRS}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$ and is parameterized by κ.

- Upon receiving (sid, i) from every party $P_i \in \mathcal{P}$, the functionality computes $\mathsf{Gen}(1^\kappa) \rightarrow (\mathsf{ck}, \tau_0, \tau_1)$ and $G(1^\kappa) \rightarrow (\mathsf{pk}, \mathsf{sk})$, where G is the key-generation algorithm of an IND-CCA secure encryption scheme[a] and Gen is the key-generation algorithm of a double-trapdoor homomorphic commitment scheme. The functionality then sets $\mathsf{CRS} = (\mathsf{ck}, \mathsf{pk})$ and sends $(\mathsf{sid}, i, \mathsf{CRS})$ to every party $P_i \in \mathcal{P}$ and the adversary $\mathcal{S}$ and halts.

[a] For use in the protocol of [12]

Fig. 2. The Ideal Functionality for Generating CRS

Functionality $\mathcal{F}_{\text{BC}}$

$\mathcal{F}_{\text{BC}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$.

- Upon receiving $(\mathsf{sid}, \mathsf{Sen}, x, \mathcal{X})$ from the sender $\mathsf{Sen} \in \mathcal{P}$ such that $\mathcal{X} \subseteq \mathcal{P}$, the functionality sends $(\mathsf{sid}, j, \mathsf{Sen}, x)$ to every $P_j \in \mathcal{X}$ and to the adversary $\mathcal{S}$ and halts.

Fig. 3. The Ideal Functionality for Broadcast

Functionality Related to $[\cdot]$-sharings: In Figure 5 we present the functionality $\mathcal{F}_{\text{GEN}[\cdot]}$ which allows a dealer $\mathsf{D} \in \mathcal{P}$ to verifiably $[\cdot]$-share an already committed secret among the parties in $\mathcal{P}$. The functionality is invoked when it receives three polynomials, say $f(\cdot), g(\cdot)$ and $h(\cdot)$ from the dealer D and a commitment, say $\mathbf{C}$, supposedly the commitment of $f(0)$ with randomness $g(0), h(0)$ (namely $\mathbf{C}_{f(0),g(0),h(0)}$), from the (majority of the) parties in $\mathcal{P}$. The functionality then hands $f_i = f(\alpha_i), g_i = g(\alpha_i), h_i = h(\alpha_i)$ and commitments $\{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}}$ to $P_i \in \mathcal{P}$ after 'verifying' that **(a):** All the three polynomials are of degree at most

Functionality $\mathcal{F}_{\text{COMMITTEE}}$

$\mathcal{F}_{\text{COMMITTEE}}$, parametrized by a constant $\mathfrak{c}$, interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$.

- Upon receiving $(\mathsf{sid}, i, \mathcal{Y})$ from every $P_i \in \mathcal{P}$, the functionality selects $\mathfrak{c}$ parties at random from the set $\mathcal{Y}$ that is received from the majority of the parties and denotes the selected set as $\mathcal{X}$. The functionality then sends $(\mathsf{sid}, i, \mathcal{X})$ to every $P_i \in \mathcal{P}$ and $\mathcal{S}$ and halts.

Fig. 4. The Ideal Functionality for Selecting a Random Committee of Given Size $\mathfrak{c}$

t and **(b):** $\mathbf{C} = \mathsf{Comm}_{\mathsf{ck}}(f(0); g(0), h(0))$ i.e. the value (and the corresponding randomness) committed in $\mathbf{C}$ are embedded in the constant term of $f(\cdot), g(\cdot)$ and $h(\cdot)$ respectively. If either of the above two checks fail, then the functionality returns Failure to the parties indicating that D is corrupted.

In our MPC protocol where $\mathcal{F}_{\text{GEN}[\cdot]}$ is called, the dealer will compute the commitment $\mathbf{C}$ as $\mathsf{Comm}_{\mathsf{ck}}(f(0); g(0), h(0))$ and will broadcast it prior to making a call to $\mathcal{F}_{\text{GEN}[\cdot]}$. It is easy to note that $\mathcal{F}_{\text{GEN}[\cdot]}$ generates $[f(0)]$ if D is honest or well-behaved. If $\mathcal{F}_{\text{GEN}[\cdot]}$ returns Failure, then D is indeed corrupted.

Functionality $\mathcal{F}_{\text{GEN}[\cdot]}$

$\mathcal{F}_{\text{GEN}[\cdot]}$ interacts with the parties in $\mathcal{P}$, a dealer $\mathsf{D} \in \mathcal{P}$, and the adversary $\mathcal{S}$ and is parametrized by a commitment key ck of a double-trapdoor homomorphic commitment scheme, along with t.

- On receiving $(\mathsf{sid}, \mathsf{D}, f(\cdot), g(\cdot), h(\cdot))$ from D and $(\mathsf{sid}, i, \mathsf{D}, \mathbf{C})$ from every $P_i \in \mathcal{P}$, the functionality verifies whether $f(\cdot), g(\cdot)$ and $h(\cdot)$ are of degree at most t and $\mathbf{C} \stackrel{?}{=} \mathsf{Comm}_{\mathsf{ck}}(f(0); g(0), h(0))$, where $\mathbf{C}$ is received from the majority of the parties.
- If any of the above verifications fail then the functionality sends $(\mathsf{sid}, i, \mathsf{D}, \mathsf{Failure})$ to every $P_i \in \mathcal{P}$ and $\mathcal{S}$ and halts.
- Else for every $P_i \in \mathcal{P}$, the functionality computes the share $f_i = f(\alpha_i)$, the opening information $g_i = g(\alpha_i), h_i = h(\alpha_i)$, and the commitment $\mathbf{C}_{f_i,g_i,h_i} = \mathsf{Comm}_{\mathsf{ck}}(f_i; g_i, h_i)$. It sends $(\mathsf{sid}, i, \mathsf{D}, [s]_i)$ to every $P_i \in \mathcal{P}$ where $[s]_i = (f_i, g_i, h_i, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}})$ and halts.

Fig. 5. The Ideal Functionality for Verifiably Generating $[\cdot]$-sharing

We note that $\mathcal{F}_{\text{GEN}[\cdot]}$ is slightly different from the standard ideal functionality (see e.g. [2]) of verifiable secret sharing (VSS) where the parties output *only* their shares (and not the commitment of all the shares). In most of the standard instantiations of a VSS functionality (in the computational setting), for example the Pedersen VSS [27], a public commitment of all the shares and the secret are available to the parties without violating any privacy. In order to make these commitments available to the external protocol that invokes $\mathcal{F}_{\text{GEN}[\cdot]}$, we allow the functionality to compute and deliver the shares along with the commitments to the parties. We note, [1] introduced a similar functionality for "committed VSS" that outputs to the parties the commitment of the secret provided by the dealer due to the same motivation mentioned above.

3.2 Supporting Sub-protocols

Our MPC protocol also makes use of the following sub-protocols. Due to space constraints, here we only present a high level description of these protocols and state their communication complexity. The formal details of the protocols are available in Appendix C. Since we later show that our main MPC protocol that invokes these sub-protocols is UC-secure, it is not required to prove any form of security for these sub-protocols separately.

(A) Protocol $\Pi_{\langle\cdot\rangle \to [\cdot]}$ (Figure 10, Appendix C) : it takes input $\langle s \rangle_{\mathcal{X}}$ for a set $\mathcal{X}$ containing at least *one* honest party and either produces a sharing $[s]$ (if all the parties in $\mathcal{X}$ behave honestly) or outputs one of

the following: the identity of a single corrupted party or a pair of parties (with at least one of them being corrupted) from $\mathcal{X}$. The protocol makes use of the functionalities $\mathcal{F}_{\text{GEN}[\cdot]}$ and $\mathcal{F}_{\text{BC}}$.

More specifically, let $\langle s \rangle_i$ denote the information (namely the share, opening information and the set of commitments) of party $P_i \in \mathcal{X}$ corresponding to the sharing $\langle s \rangle_{\mathcal{X}}$. To achieve the goal of our protocol, there are two clear steps to perform: *first*, the correct commitment for each share of s corresponding to its $\langle \cdot \rangle_{\mathcal{X}}$-sharing, now available to the parties in $\mathcal{X}$, is to be made available to *all* the parties in $\mathcal{P}$; *second*, each $P_i \in \mathcal{X}$ is required to act as a dealer and verifiably $[\cdot]$-share its already committed share s_i among $\mathcal{P}$. Note that the commitment to s_i is included in the set of commitments that will be already available among $\mathcal{P}$ due to the first step. Clearly, once $[s_i]$ are generated for each $P_i \in \mathcal{X}$, then $[s]$ is computed as $[s] = \sum_{P_i \in \mathcal{X}} [s_i]$; this is because $s = \sum_{P_i \in \mathcal{X}} s_i$.

Now there are two steps that may lead to the failure of the protocol. First, $P_i \in \mathcal{X}$ may be identified as a corrupted dealer while calling $\mathcal{F}_{\text{GEN}[\cdot]}$. In this case a single corrupted party is outputted by every party in $\mathcal{P}$. Second, the protocol may fail when the parties in $\mathcal{P}$ try to reach an agreement over the correct set of commitments of the shares of s. Recall that each $P_i \in \mathcal{X}$ holds a set of commitments as a part of $\langle s \rangle_{\mathcal{X}}$. We ask each $P_i \in \mathcal{X}$ to call $\mathcal{F}_{\text{BC}}$ to broadcast among $\mathcal{P}$ the set of commitments held by him. It is necessary to ask each $P_i \in \mathcal{X}$ to do this as we can not trust any single party from $\mathcal{X}$, since all we know (with overwhelming probability) is that $\mathcal{X}$ contains at least one honest party. Now if the parties in $\mathcal{P}$ receive the same set of commitments from all the parties in $\mathcal{X}$, then clearly the received set is the correct set of commitments and agreement on the set is reached among $\mathcal{P}$. If this does not happen the parties in $\mathcal{P}$ can detect a pair of parties with conflicting sets and output the said pair. It is not hard to see that indeed one party in the pair must be corrupted. To ensure an agreement on the selected pair when there are multiple such conflicting pairs, we assume the existence of a predefined publicly known algorithm to select a pair from the lot (for instance consider the pair (P_a, P_b) with minimum value of $a + n \cdot b$). Intuitively the protocol is secure as the shares of honest parties in $\mathcal{X}$ remain secure.

The communication complexity of protocol $\Pi_{\langle \cdot \rangle \to [\cdot]}$ is stated in Lemma 1, which easily follows from the fact that each party in $\mathcal{X}$ needs to broadcast $\mathcal{O}(|\mathcal{X}|\kappa)$ bits to $\mathcal{P}$.

Lemma 1. *The communication complexity of protocol $\Pi_{\langle \cdot \rangle \to [\cdot]}$ is $\mathcal{BC}\big(|\mathcal{X}|^2\kappa, n\big)$ plus the complexity of $\mathcal{O}(|\mathcal{X}|)$ invocations to the realization of the functionality $\mathcal{F}_{\text{GEN}[\cdot]}$.*

(B) Protocol $\Pi_{\langle \cdot \rangle}$ (Figure 11, Appendix C) : the protocol enables a designated party (dealer) $\mathsf{D} \in \mathcal{P}$ to verifiably $\langle \cdot \rangle$-share an already committed secret f among a set of parties $\mathcal{X}$ containing at least one honest party. More specifically, every $P_i \in \mathcal{P}$ holds a (publicly known) commitment $\mathbf{C}_{f,g,h}$. The dealer D holds the secret $f \in \mathbb{F}_p$ and randomness pair (g, h), such that $\mathbf{C}_{f,g,h} = \mathsf{Comm_{ck}}(f; g, h)$; and the goal is to generate $\langle f \rangle_{\mathcal{X}}$. In the protocol, D first additively shares f as well as the opening information (g, h) among $\mathcal{X}$. In addition, D is also asked to publicly commit each additive-share of f, using the corresponding additive-share of (g, f). The parties can then publicly verify whether indeed D has $\langle \cdot \rangle$-shared the same f as committed in $\mathbf{C}_{f,g,h}$, via the homomorphic property of the commitments. Intuitively f remains private in the protocol for an honest D as there exists at least one honest party in $\mathcal{X}$. Moreover the binding property of the commitment ensures that a potentially corrupted D fails to $\langle \cdot \rangle$-share an incorrect value $f' \neq f$.

If we notice carefully the protocol achieves a little more than $\langle \cdot \rangle$-sharing of a secret among a set of parties $\mathcal{X}$. All the parties in $\mathcal{P}$ hold the commitments to the shares of f, while as per the definition of $\langle \cdot \rangle$-sharing the commitments to shares should be available to the parties in $\mathcal{X}$ alone. A closer look reveals that the public commitments to the shares of f among the parties in $\mathcal{P}$ enable them to publicly verify whether D has indeed $\langle \cdot \rangle$-shared the same f among $\mathcal{X}$ as committed in $\mathbf{C}_{f,g,h}$ via the homomorphic property of the commitments. The communication complexity of $\Pi_{\langle \cdot \rangle}$ is stated in Lemma 2.

Lemma 2. *The communication complexity of protocol $\Pi_{\langle\cdot\rangle}$ is $\mathcal{O}(|\mathcal{X}|\kappa)$ and $\mathcal{BC}\big(|\mathcal{X}|\kappa, n\big)$.*

(C) Protocol $\Pi_{[\cdot]\to\langle\cdot\rangle}$ (Figure 12, Appendix C): the protocol takes as input $[s]$ for any secret s and outputs $\langle s\rangle_{\mathcal{X}}$ for a designated set of parties $\mathcal{X} \subset \mathcal{P}$ containing at least one honest party.

Let $f_1, \ldots, f_n$ be the Shamir-shares of s. Then the protocol is designed using the following two-stage approach: **(1)**: First each party $P_k \in \mathcal{P}$ acts as a dealer and verifiably $\langle\cdot\rangle$-share's its share f_k via protocol $\Pi_{\langle\cdot\rangle}$; **(2)** Let $\mathcal{H}$ be the set of $|\mathcal{H}| > t+1$ parties P_k who have correctly $\langle\cdot\rangle$-shared its Shamir-share f_k; without loss of generality, let $\mathcal{H}$ be the set of first $|\mathcal{H}|$ parties in $\mathcal{P}$. Since the original sharing polynomial (for $[\cdot]$-sharing s) has degree at most t with s as its constant term, then there exists publicly known constants (namely the Lagrange's interpolation coefficients) $c_1, \ldots, c_{|\mathcal{H}|}$, such that $s = c_1 f_1 + \ldots + c_{|\mathcal{H}|} f_{|\mathcal{H}|}$. Since corresponding to each $P_k \in \mathcal{H}$ the share f_k is $\langle\cdot\rangle$-shared, it follows easily that each party $P_i \in \mathcal{X}$ can compute $\langle s\rangle_i = c_1\langle f_1\rangle_i + \ldots + c_{|\mathcal{H}|}\langle f_{|\mathcal{H}|}\rangle_i$. The correctness of the protocol follows from the fact that the corrupted parties in $\mathcal{P}$ will fail to $\langle\cdot\rangle$-share an incorrect Shamir-share of s, thanks to the protocol $\Pi_{\langle\cdot\rangle}$. The privacy of s follows from the fact that the Shamir shares of the honest parties in $\mathcal{P}$ remain private, which follows from the privacy of the protocol $\Pi_{\langle\cdot\rangle}$.

The communication complexity of the protocol $\Pi_{[\cdot]\to\langle\cdot\rangle}$ is stated in Lemma 3 which follows from the fact that n invocations to $\Pi_{\langle\cdot\rangle}$ are done in the protocol.

Lemma 3. *The communication complexity of $\Pi_{[\cdot]\to\langle\cdot\rangle}$ is $\mathcal{O}(n|\mathcal{X}|\kappa)$ and $\mathcal{BC}\big(n|\mathcal{X}|\kappa, n\big)$.*

(D) Protocol $\Pi_{\textbf{RANDZERO}[\cdot]}$ (Figure 14, Appendix C): the protocol is used for generating a random $[\cdot]$-sharing of 0. To design the protocol, we also require a standard Zero-knowledge (ZK) functionality $\mathcal{F}_{\text{ZK.BC}}$ to publicly prove a commitment to zero. The functionality is a "prove-and-broadcast " functionality that upon receiving a commitment and witness pair $(\mathbf{C}, (u, v))$ from a designated prover P_j, verifies if $\mathbf{C} = \mathsf{Comm}_{\mathsf{ck}}(0; u, v)$ or not. If so it sends $\mathbf{C}$ to all the parties. A protocol $\Pi_{\text{ZK.BC}}$ realizing $\mathcal{F}_{\text{ZK.BC}}$ can be designed in the CRS model using standard techniques, see [22], with communication complexity $\mathcal{O}(\mathsf{Poly}(n)\kappa)$.

Protocol $\Pi_{\text{RANDZERO}[\cdot]}$ invokes the ideal functionalities $\mathcal{F}_{\text{ZK.BC}}$ and $\mathcal{F}_{\text{GEN}[\cdot]}$. The idea is as follows: Each party $P_i \in \mathcal{P}$ first broadcasts a random commitment of 0 and proves in a zero-knowledge (ZK) fashion that it indeed committed 0. Next P_i calls $\mathcal{F}_{\text{GEN}[\cdot]}$ as a dealer D to generate $[\cdot]$-sharing of 0 that is consistent with the commitment of 0. The parties then locally add the sharings of the dealers who are successful as dealers in their corresponding calls to $\mathcal{F}_{\text{GEN}[\cdot]}$. Since there exists at least one honest party in this set of dealers, the resultant sharing will be indeed a random sharing of 0, see Appendix C for details. Looking ahead, we invoke $\Pi_{\text{RANDZERO}[\cdot]}$ only once in our main MPC protocol Π_f (more on this later); so we avoid giving details of the communication complexity of the protocol. However assuming standard realization of $\mathcal{F}_{\text{ZK.BC}}$, the protocol has complexity $\mathcal{O}(\mathsf{Poly}(n)\kappa)$.

(E) Dis-honest Majority MPC Protocol (Appendix D): Apart from the above sub-protocols, we use a non-robust, dishonest-majority MPC protocol $\Pi_{\mathsf{C}}^{\text{NR}}$ with the capability of fault-detection. The protocol, presented in Figure 18 of Appendix D, allows a designated set of parties $\mathcal{X} \subset \mathcal{P}$, containing at least one honest party, to perform $\langle\cdot\rangle$-shared evaluation of a given circuit C. In case some corrupted party in $\mathcal{X}$ behaves maliciously, the parties in $\mathcal{P}$ identify a pair of parties from $\mathcal{X}$, with at least one of them being corrupted. The starting point of $\Pi_{\mathsf{C}}^{\text{NR}}$ is the dishonest majority MPC protocol of [12], which takes $\langle\cdot\rangle$-shared inputs of a given circuit, from a set of parties, say $\mathcal{X}$, having a dishonest majority. The protocol then achieves the following:

- If all the parties in $\mathcal{X}$ behave honestly, then the protocol outputs $\langle\cdot\rangle$-shared circuit outputs among $\mathcal{X}$.
- Else the honest parties in $\mathcal{X}$ detect misbehaviour by the corrupted parties and abort the protocol.

We observe that for an aborted execution of the protocol of [12], there exists an honest party in $\mathcal{X}$ that can *locally* identify a corrupted party from $\mathcal{X}$, who deviated from the protocol. We exploit this property in $\Pi_{\mathsf{C}}^{\mathrm{NR}}$ to enable the parties in $\mathcal{P}$ identify a pair of parties from $\mathcal{X}$ with at least one of them being corrupted.

Protocol $\Pi_{\mathsf{C}}^{\mathrm{NR}}$ proceeds in two stages, the *preparation stage* and the *evaluation stage*, each involving various other sub-protocols (details available in Appendix D). In the preparation stage, if all the parties in $\mathcal{X}$ behave honestly, then they jointly generate $\mathsf{C_M} + \mathsf{C_R}$ shared multiplication triples $\{(\langle \mathbf{a}^{(i)} \rangle_{\mathcal{X}}, \langle \mathbf{b}^{(i)} \rangle_{\mathcal{X}}, \langle \mathbf{c}^{(i)} \rangle_{\mathcal{X}})\}_{i=1,\ldots,\mathsf{C_M}+\mathsf{C_R}}$, such that $\mathbf{c}^{(i)} = \mathbf{a}^{(i)} \cdot \mathbf{b}^{(i)}$ and each $(\mathbf{a}^{(i)}, \mathbf{b}^{(i)}, \mathbf{c}^{(i)})$ is random and unknown to the adversary; here $\mathsf{C_M}$ and $\mathsf{C_R}$ are the number of multiplication and random gates in C respectively. Otherwise, the parties in $\mathcal{P}$ identify a pair of parties in $\mathcal{X}$, with at least one of them being corrupted.

Assuming that the desired $\langle \cdot \rangle$-shared multiplication triples are generated in the preparation stage, the parties in $\mathcal{X}$ start evaluating C in a shared fashion by maintaining the following standard invariant for each gate of C: *Given $\langle \cdot \rangle$-shared inputs of the gate, the parties securely compute the $\langle \cdot \rangle$-shared output of the gate*. Maintaining the invariant for the linear gates in C does not require any interaction, thanks to the linearity of $\langle \cdot \rangle$-sharing. For a multiplication gate, the parties deploy a preprocessed $\langle \cdot \rangle$-shared multiplication triple from the preparation stage (for each multiplication gate a different triple is deployed) and use the standard Beaver's trick [3], (see protocol Π_{BEA} in Appendix D) . While applying Beaver's trick, the parties in $\mathcal{X}$ need to publicly open two $\langle \cdot \rangle$-shared values using a reconstruction protocol $\Pi_{\mathrm{REC}\langle \cdot \rangle}$ (presented in Appendix D). It may be possible that the opening is non-robust[8], in which case the circuit evaluation fails and the parties in $\mathcal{P}$ identify a pair of parties from $\mathcal{X}$ with at least one of them being corrupted. For a random gate, the parties consider an $\langle \cdot \rangle$-shared multiplication triple from the preparation stage (for each random gate a different triple is deployed) and the first component of the triple is considered as the output of the random gate. The protocol ends once the parties in $\mathcal{X}$ obtain $\langle \cdot \rangle$-shared circuit outputs $\langle y_1 \rangle_{\mathcal{X}}, \ldots, \langle y_{\mathsf{out}} \rangle_{\mathcal{X}}$; so no reconstruction is required at the end.

The complete details of $\Pi_{\mathsf{C}}^{\mathrm{NR}}$ is provided in Appendix D. The protocol invokes two ideal functionalities $\mathcal{F}_{\mathrm{GENRAND}\langle \cdot \rangle}$ and $\mathcal{F}_{\mathrm{BC}}$ where the functionality $\mathcal{F}_{\mathrm{GENRAND}\langle \cdot \rangle}$ is used to generate $\langle \cdot \rangle$-sharing of random values (again see Appendix D). For our purpose we note that the protocol provides a statistical security of 2^{-s} and has communication complexity as stated in Lemma 4 and proved in Appendix D. Note that there are two types of broadcast involved: among the parties in $\mathcal{X}$ and among the parties in $\mathcal{P}$.

Lemma 4. *For a statistical security parameter s, protocol $\Pi_{\mathsf{C}}^{\mathrm{NR}}$ has communication complexity of $\mathcal{O}(|\mathcal{X}|^2(|\mathsf{C}|+s)\kappa), \mathcal{BC}\big(|\mathcal{X}|^2(|\mathsf{C}|+s)\kappa, |\mathcal{X}|\big)$ and $\mathcal{BC}\big(|\mathcal{X}|\kappa, n\big)$.*

3.3 The MPC Protocol

Finally, we describe our MPC protocol. Recall that we divide the circuit ckt into sub-circuits $\mathsf{ckt}_1, \ldots, \mathsf{ckt}_L$ and we let in_l and out_l denote the number of input and output wires respectively for the sub-circuit ckt_l. At the beginning of the protocol, each party $[\cdot]$-share their private inputs by calling $\mathcal{F}_{\mathrm{GEN}[\cdot]}$. The parties then select a random committee of parties by calling $\mathcal{F}_{\mathrm{COMMITTEE}}$ for evaluating the lth sub-circuit via the dishonest majority MPC protocol of [12]. We use a Boolean flag NewCom in the protocol to indicate if a new committee has to be decided, prior to the evaluation of lth sub-circuit or the committee used for the evaluation of the $(l-1)$th sub-circuit is to be continued. Specifically a successful evaluation of a sub-circuit is followed by setting NewCom equals to 0, implying that the current committee is to be continued for the evaluation of the subsequent sub-circuit. On the other hand, a failed evaluation of a sub-circuit is followed by setting NewCom equals to 1, implying that a fresh committee has to be decided for the re-evaluation of the same sub-circuit from the updated set of eligible parties $\mathcal{L}$, which is modified after the failed evaluation.

[8] As we may not have honest majority in $\mathcal{X}$, we could not always ensure robust reconstruction during $\Pi_{\mathrm{REC}\langle \cdot \rangle}$.

After each successful sub-circuit evaluation, the corresponding $\langle\cdot\rangle$-shared outputs are converted into $[\cdot]$-shared outputs via protocol $\Pi_{\langle\cdot\rangle\to[\cdot]}$, while prior to each sub-circuit evaluation, the corresponding $[\cdot]$-shared inputs are converted to the required $\langle\cdot\rangle$-shared inputs via protocol $\Pi_{[\cdot]\to\langle\cdot\rangle}$. The process is repeated till the function output is $[\cdot]$-shared, after which it is robustly reconstructed (as we have honest majority in $\mathcal{P}$).

Without affecting the correctness of the above steps, but to ensure simulation security (in the UC model), we add an additional output re-randomization step before the output reconstruction: the parties call $\Pi_{\text{RANDZERO}[\cdot]}$ to generate a random $[0]$, which they add to the $[\cdot]$-shared output (thus keeping the same function output). Looking ahead, during the simulation in the security proof, this step allows the simulator to cheat and set the final output to be the one obtained from the functionality, even though it simulates the honest parties with 0 as the input (see Appendix E for the details).

Let E be the event that at least one party in each of the selected committees during sub-circuit evaluations is honest; the event E occurs except with probability at most $(t+1)\cdot\epsilon^{\mathfrak{c}} \approx 2^{-\kappa}$. This is because at most $(t+1)$ (random) committees need to be selected (a new committee is selected after each of the t failed sub-circuit evaluation plus an initial selection is made). It is easy to see that conditioned on E, the protocol is private: the inputs of the honest parties remain private during the input stage (due to $\mathcal{F}_{\text{GEN}[\cdot]}$), while each of the involved sub-protocols for sub-circuit evaluations does not leak any information about honest party's inputs. It also follows that conditioned on E, the protocol is correct, thanks to the binding property of the commitment and the properties of the involved sub-protocols.

The properties of the protocol Π_f are stated in Theorem 1 and the security proof is available in Appendix E; we only provide the proof of communication complexity here. The (circuit-dependent) communication complexity in the theorem is derived after substituting the calls to the various ideal functionalities by the corresponding protocols implementing them. The broadcast complexity has two parts: the broadcasts among the parties in $\mathcal{P}$ and the broadcasts among small committees.

Theorem 1. *Let $f : \mathbb{F}_p^n \to \mathbb{F}_p$ be a publicly known n-input function with circuit representation ckt over $\mathbb{F}_p$, with average width w and depth d (thus $w = \frac{|\mathsf{ckt}|}{d}$). Moreover, let ckt be divided into sub-circuits $\mathsf{ckt}_1, \ldots, \mathsf{ckt}_L$, with $L = t$ and each sub-circuit ckt_l having fan-in in_l and fan-out out_l. Furthermore, let $\mathsf{in}_l = \mathsf{out}_l = \mathcal{O}(w)$. Then conditioned on the event E, protocol Π_f (κ, s)-securely realizes the functionality $\mathcal{F}_f$ against $\mathcal{A}$ in the $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{COMMITTEE}}, \mathcal{F}_{\text{GEN}[\cdot]}, \mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}, \mathcal{F}_{\text{ZK.BC}})$-hybrid model[9] in the UC security framework. The circuit-dependent communication complexity of the protocol is $\mathcal{O}(|\mathsf{ckt}|\cdot(\frac{n\cdot t}{d}+\kappa)\cdot\kappa^2)$, $\mathcal{BC}\big(|\mathsf{ckt}|\cdot\frac{n\cdot t\cdot\kappa^2}{d}, n\big)$ and $\mathcal{BC}\big(|\mathsf{ckt}|\cdot\kappa^3, \kappa\big)$.*

PROOF (COMMUNICATION COMPLEXITY): We analyze each phase of the protocol:

1. **Input Commitment Stage:** Here each party broadcasts $\mathcal{O}(\kappa)$ bits to the parties in $\mathcal{P}$ and so the broadcast complexity of this step is $\mathcal{BC}\big(n\kappa, n\big)$.
2. **$[\cdot]$-sharing of Committed Inputs:** Here n calls to $\mathcal{F}_{\text{GEN}[\cdot]}$ are made. Realizing $\mathcal{F}_{\text{GEN}[\cdot]}$ with the protocol $\Pi_{[\cdot]}$, see Lemma 5, this incurs a communication complexity of $\mathcal{O}(n^2\kappa)$ and $\mathcal{BC}\big(n^2\kappa, n\big)$.
3. **Sub-circuit Evaluations:** We first count the total communication cost of evaluating the sub-circuit ckt_l with in_l input gates and out_l output gates.
   - Converting the in_l $[\cdot]$-shared inputs to in_l $\langle\cdot\rangle$-shared inputs will require in_l invocations to the protocol $\Pi_{[\cdot]\to\langle\cdot\rangle}$. The communication complexity of this step is $\mathcal{O}(n\cdot\mathfrak{c}\cdot\mathsf{in}_l\cdot\kappa)$ and $\mathcal{BC}\big(n\cdot\mathfrak{c}\cdot\mathsf{in}_l\cdot\kappa, n\big)$; this follows from Lemma 3 by substituting $|\mathcal{X}| = \mathfrak{c}$.
   - Since the size of ckt_l is at most $\frac{|\mathsf{ckt}|}{L}$, evaluating the same via protocol $\Pi^{\mathsf{NR}}_{\mathsf{ckt}_l}$ will have communication complexity $\mathcal{O}(\mathfrak{c}^2(\frac{|\mathsf{ckt}|}{L}+s)\kappa)$, $\mathcal{BC}\big(\mathfrak{c}^2(\frac{|\mathsf{ckt}|}{L}+s)\kappa, \mathfrak{c}\big)$ and $\mathcal{BC}\big(\mathfrak{c}\cdot\kappa, n\big)$; this follows from Lemma 4 by substituting $|\mathcal{X}| = \mathfrak{c}$.

[9] See Appendix A for the meaning of g-hybrid model in the UC framework.

Protocol $\Pi_f(\mathcal{P}, \mathsf{ckt})$

For session ID sid, every party $P_i \in \mathcal{P}$ does the following:

Initialization. Set $\mathcal{L} = \mathcal{P}$, $\mathfrak{n} = |\mathcal{L}|$, $\mathfrak{t} = t$ and $\mathsf{NewCom} = 1$. Divide ckt into L sub-circuits $\mathsf{ckt}_1, \ldots, \mathsf{ckt}_L$, each of depth d/L.

CRS Generation. Invoke $\mathcal{F}_{\text{CRS}}$ with (sid, i) and get back $(\mathsf{sid}, i, \mathsf{CRS})$, where $\mathsf{CRS} = (\mathsf{pk}, \mathsf{ck})$.

Input Commitment. On input $x^{(i)}$, choose random polynomials $f^{(i)}(\cdot), g^{(i)}(\cdot), h^{(i)}(\cdot)$ of degree $\leq t$, such that $f^{(i)}(0) = x^{(i)}$ and compute the commitment $\mathbf{C}_{x^{(i)}, g_0^{(i)}, h_0^{(i)}} = \mathsf{Comm}_{\mathsf{ck}}(x^{(i)}; g_0^{(i)}, h_0^{(i)})$ where $g_0^{(i)} = g^{(i)}(0),\ h_0^{(i)} = h^{(i)}(0)$.

- Call $\mathcal{F}_{\text{BC}}$ with message $(\mathsf{sid}, i, \mathbf{C}_{x^{(i)}, g_0^{(i)}, h_0^{(i)}}, \mathcal{P})$.
- Corresponding to each $P_j \in \mathcal{P}$, receive $(\mathsf{sid}, i, j, \mathbf{C}_{x^{(j)}, g_0^{(j)}, h_0^{(j)}})$ from $\mathcal{F}_{\text{BC}}$.

$[\cdot]$-sharing of Committed Inputs.

- Act as a dealer D and call $\mathcal{F}_{\text{GEN}[\cdot]}$ with $(\mathsf{sid}, i, f^{(i)}(\cdot), g^{(i)}(\cdot), h^{(i)}(\cdot))$.
- For every $P_j \in \mathcal{P}$, call $\mathcal{F}_{\text{GEN}[\cdot]}$ with $(\mathsf{sid}, i, j, \mathbf{C}_{x^{(j)}, g_0^{(j)}, h_0^{(j)}})$.
- For every $P_j \in \mathcal{P}$, if $(\mathsf{sid}, i, j, \mathsf{Failure})$ is received from $\mathcal{F}_{\text{GEN}[\cdot]}$, substitute a default predefined public sharing $[0]$ of 0 as $[x^{(j)}]$, set $[x^{(j)}]_i = [0]_i$ and update $\mathcal{L} = \mathcal{L} \setminus \{P_j\}$, decrement $\mathfrak{t}$ and $\mathfrak{n}$ by one. Else receive $(\mathsf{sid}, i, j, [x^{(j)}]_i)$ from $\mathcal{F}_{\text{GEN}[\cdot]}$.

Start of While Loop Over the Sub-circuits. Initialize $l = 1$. While $l \leq L$ do:

– Committee Selection. If $\mathsf{NewCom} = 1$, then call $\mathcal{F}_{\text{COMMITTEE}}$ with $(\mathsf{sid}, i, \mathcal{L})$ and receive $(\mathsf{sid}, i, \mathcal{C})$ from $\mathcal{F}_{\text{COMMITTEE}}$.

– $[\cdot]$ to $\langle\cdot\rangle_{\mathcal{C}}$ Conversion of Inputs of Sub-circuit ckt_l. Let $[x_1], \ldots, [x_{\mathsf{in}_l}]$ denote the $[\cdot]$-sharing of the inputs to ckt_l:

- For $k = 1, \ldots, \mathsf{in}_l$, participate in $\Pi_{[\cdot]\to\langle\cdot\rangle}$ with $(\mathsf{sid}, i, [x_k]_i, \mathcal{C})$. Output $(\mathsf{sid}, i, \langle x_k\rangle_i)$ in $\Pi_{[\cdot]\to\langle\cdot\rangle}$, if P_i belongs to $\mathcal{C}$. Else output (sid, i).

– Evaluation of the Sub-circuit ckt_l. If $P_i \in \mathcal{C}$ then participate in $\Pi^{\text{NR}}_{\mathsf{ckt}_l}$ with $(\mathsf{sid}, i, \langle x_1\rangle_i, \ldots, \langle x_{\mathsf{in}_l}\rangle_i, \mathcal{C})$, else participate in $\Pi^{\text{NR}}_{\mathsf{ckt}_l}$ with $(\mathsf{sid}, i, \mathcal{C})$.

- If $(\mathsf{sid}, i, \mathsf{Failure}, P_a, P_b)$ is the output during $\Pi^{\text{NR}}_{\mathsf{ckt}_l}$, then set $\mathcal{L} = \mathcal{L} \setminus \{P_a, P_b\}$, $\mathfrak{t} = \mathfrak{t} - 1$, $\mathfrak{n} = \mathfrak{n} - 2$, $\mathsf{NewCom} = 1$ and go to **Committee Selection** step.

– $\langle\cdot\rangle_{\mathcal{C}}$ to $[\cdot]$ conversion of Outputs of ckt_l. If $(\mathsf{sid}, i, \mathsf{Success}, \langle y_1\rangle_i, \ldots, \langle y_{\mathsf{out}_l}\rangle_i)$ or $(\mathsf{sid}, i, \mathsf{Success})$ is obtained during $\Pi^{\text{NR}}_{\mathsf{ckt}_l}$, then participate in $\Pi_{\langle\cdot\rangle\to[\cdot]}$ with $(\mathsf{sid}, i, \langle y_k\rangle_i)$ or (sid, i) (respectively) for $k = 1, \ldots, \mathsf{out}_l$.

- If $(\mathsf{sid}, i, \mathsf{Success}, [y_k]_i)$ is the output in $\Pi_{\langle\cdot\rangle\to[\cdot]}$ for every $k = 1, \ldots, \mathsf{out}_l$, then increment l and set $\mathsf{NewCom} = 0$.
- If $(\mathsf{sid}, i, \mathsf{Failure}, P_a, P_b)$ is the output in $\Pi_{\langle\cdot\rangle\to[\cdot]}$ for *some* $k \in \{1, \ldots, \mathsf{out}_l\}$, then set $\mathcal{L} = \mathcal{L} \setminus \{P_a, P_b\}$, $\mathfrak{t} = \mathfrak{t} - 1$, $\mathfrak{n} = \mathfrak{n} - 2$, $\mathsf{NewCom} = 1$ and go to the **Committee Selection** step.
- If $(\mathsf{sid}, i, \mathsf{Failure}, P_a)$ is the output in $\Pi_{\langle\cdot\rangle\to[\cdot]}$ for *some* $k \in \{1, \ldots, \mathsf{out}_l\}$, then set $\mathcal{L} = \mathcal{L} \setminus \{P_a\}$, $\mathfrak{t} = \mathfrak{t} - 1$, $\mathfrak{n} = \mathfrak{n} - 1$, $\mathsf{NewCom} = 1$ and go to the **Committee Selection** step.

Output Rerandomization. Let $[y]$ denote the $[\cdot]$-sharing of the output of ckt. Participate in $\Pi_{\text{RANDZERO}[\cdot]}$ with (sid, i), obtain $(\mathsf{sid}, i, [0]_i)$ and locally compute $[z]_i = [y]_i + [0]_i$.

Output Reconstruction. Interpret $[z]_i$ as $(f_i, g_i, h_i, \{\mathbf{C}_{f_j, g_j, h_j}\}_{P_j \in \mathcal{P}})$. Initialize a set $\mathcal{T}_i$ to $\emptyset$.

- Send $(\mathsf{sid}, i, j, f_i, g_i, h_i)$ to every $P_j \in \mathcal{P}$. On receiving $(\mathsf{sid}, j, i, f_j, g_j, h_j)$ from every party P_j include party P_j in $\mathcal{T}_i$ if $\mathbf{C}_{f_j, g_j, h_j} \neq \mathsf{Comm}_{\mathsf{ck}}(f_j; (g_j, h_j))$.
- Interpolate $f(\cdot)$ such that $f(\alpha_j) = f_j$ holds for every $P_j \in \mathcal{P} \setminus \mathcal{T}_i$. If $f(\cdot)$ has degree at most t, output $(\mathsf{sid}, i, z = f(0))$ and halt; else output $(\mathsf{sid}, i, \mathsf{Failure})$ and halt.

Fig. 6. Protocol for UC-secure realizing $\mathcal{F}_f$

– Finally converting the out_l $\langle\cdot\rangle$-shared outputs to $[\cdot]$-shared outputs require out_l invocations to the protocol $\Pi_{\langle\cdot\rangle\to[\cdot]}$. This has communication complexity $\mathcal{O}(n \cdot \mathfrak{c} \cdot \mathsf{out}_l \cdot \kappa)$, $\mathcal{BC}\big(\mathsf{out}_l \cdot \mathfrak{c}^2 \cdot \kappa, n\big)$ and $\mathcal{BC}\big(n \cdot \mathfrak{c} \cdot \kappa, n\big)$; this follows from Lemma 1 by substituting $|\mathcal{X}| = \mathfrak{c}$.

Thus evaluating ckt_l has communication complexity $\mathcal{O}((n^2 + n \cdot \mathfrak{c} \cdot \mathsf{in}_l + n \cdot \mathfrak{c} \cdot \mathsf{out}_l + \mathfrak{c}^2(\frac{|\mathsf{ckt}|}{L} + s))\kappa)$, $\mathcal{BC}\big((n^2 + n \cdot \mathfrak{c} \cdot \mathsf{in}_l + \mathfrak{c}^2 \cdot \mathsf{out}_l)\kappa, n\big)$ and $\mathcal{BC}\big(\mathfrak{c}^2(\frac{|\mathsf{ckt}|}{L} + s)\kappa, \mathfrak{c}\big)$. Assuming $\mathsf{in}_l = \mathcal{O}(w)$ and $\mathsf{out}_l = \mathcal{O}(w)$, with $w = \frac{|\mathsf{ckt}|}{d}$, this results in $\mathcal{O}((n^2 + n \cdot \mathfrak{c} \cdot \frac{|\mathsf{ckt}|}{d} + \mathfrak{c}^2(\frac{|\mathsf{ckt}|}{L} + s))\kappa)$, $\mathcal{BC}\big((n^2 + n \cdot \mathfrak{c} \cdot \frac{|\mathsf{ckt}|}{d})\kappa, n\big)$ and $\mathcal{BC}\big((\mathfrak{c}^2 \cdot (\frac{|\mathsf{ckt}|}{L} + s))\kappa, \mathfrak{c}\big)$. The total number of sub-circuit evaluations is at most $L + t$, with L successful evaluations and at most t failed evaluations. Substituting $L = t$, we get the overall communication complexity $\mathcal{O}((|\mathsf{ckt}| \cdot (\frac{n \cdot t \cdot \mathfrak{c}}{d} + \mathfrak{c}^2) + n^2 t + \mathfrak{c}^2 s \cdot t)\kappa)$, $\mathcal{BC}\big((|\mathsf{ckt}| \cdot \frac{n \cdot t \cdot \mathfrak{c}}{d} + n^2 t)\kappa, n\big)$ and $\mathcal{BC}\big((|\mathsf{ckt}| \cdot \mathfrak{c}^2 + \mathfrak{c}^2 \cdot s \cdot t)\kappa, \mathfrak{c}\big)$.

4. **Output Rerandomization and Reconstruction:** The costs $\mathcal{O}(\mathsf{Poly}(n, \kappa))$ bits.

The circuit-dependent complexity of the whole protocol comes out to be $\mathcal{O}(|\mathsf{ckt}| \cdot (\frac{nt \cdot \mathfrak{c}}{d} + \mathfrak{c}^2)\kappa)$ bits of communication over the point-to-point channels and broadcast-complexity of $\mathcal{BC}\big(|\mathsf{ckt}| \cdot \frac{nt \cdot \mathfrak{c}}{d} \cdot \kappa, n\big)$ and $\mathcal{BC}\big(|\mathsf{ckt}| \cdot \mathfrak{c}^2 \cdot \kappa, \mathfrak{c}\big)$. Since $\mathfrak{c}$ has to be selected so that $\epsilon^{\mathfrak{c}} < 2^{-\kappa}$ holds, asymptotically we can set $\mathfrak{c}$ to be $\mathcal{O}(\kappa)$. (For any practical purpose, $\kappa = 80$ is good enough.) It implies that the (circuit-dependent) communication complexity is $\mathcal{O}(|\mathsf{ckt}|(\frac{nt}{d} + \kappa)\kappa^2)$, $\mathcal{BC}\big(|\mathsf{ckt}| \cdot \frac{nt\kappa^2}{d}, n\big)$ and $\mathcal{BC}\big(|\mathsf{ckt}|\kappa^3, \kappa\big)$. □

We propose two optimizations for our MPC protocol that improves its communication complexity.

$[\cdot]$-sharing among a smaller subset of $\mathcal{P}$. While for simplicity, we involve the entire set of parties in $\mathcal{P}$ to hold $[\cdot]$-shared values in the protocol, it is enough to fix and involve a set of just z parties that guarantees a honest majority with overwhelming probability. From our analysis in Section 1, we find that $z = \mathcal{O}(\kappa)$. Indeed it is easy to note that all we require from the set involved in holding a $[\cdot]$-sharing is honest majority that can be attained by any set containing $\mathcal{O}(\kappa)$ parties. This optimization replaces n by κ in the complexity expressions mentioned in Theorem 1. It implies that the (circuit-dependent) communication complexity is $\mathcal{O}(|\mathsf{ckt}|(\frac{\kappa t}{d} + \kappa)\kappa^2)$, $\mathcal{BC}\big(|\mathsf{ckt}| \cdot \frac{t\kappa^3}{d}, \kappa\big)$ and $\mathcal{BC}\big(|\mathsf{ckt}|\kappa^3, \kappa\big)$. Now instantiating the broadcast functionality in the above modified protocol with the Dolev-Strong (DS) broadcast protocol (see Appendix B), we obtain the following:

Corollary 1. *If $d = \omega(t)$ and if the calls to $\mathcal{F}_{\mathrm{BC}}$ are realized via the DS broadcast protocol, then the circuit-dependent communication complexity of Π_f is $\mathcal{O}(|\mathsf{ckt}| \cdot \kappa^7)$.*

When we restrict to widths w of the form $w = \omega(\kappa^3)$, we can instantiate all the invocations to $\mathcal{F}_{\mathrm{BC}}$ in the protocols $\Pi_{\langle\cdot\rangle\to[\cdot]}$ and $\Pi_{[\cdot]\to\langle\cdot\rangle}$ (invoked before and after the sub-circuit evaluations) by the Fitzi-Hirt (FH) multi-valued broadcast protocol [19], see Appendix B. This is because, setting $w = \omega(\kappa^3)$ ensures that the combined message over all the instances of $\Pi_{\langle\cdot\rangle\to[\cdot]}$ (respectively $\Pi_{[\cdot]\to\langle\cdot\rangle}$) to be broadcast by any party satisfies the bound on the message size of the FH protocol. Incorporating the above, we obtain the following corollary with better result.

Corollary 2. *If $d = \omega(t)$ and $w = \omega(\kappa^3)$ (i.e. $|\mathsf{ckt}| = \omega(\kappa^3 t)$), then the circuit-dependent communication complexity of Π_f is $\mathcal{O}(|\mathsf{ckt}| \cdot \kappa^4)$.*

Packed Secret-Sharing. We can employ packed secret-sharing technique of [20] to checkpoint multiple outputs of the sub-circuits together in a single $[\cdot]$-sharing. Specifically, if we involve all the parties in $\mathcal{P}$ to hold a $[\cdot]$-sharing, we can pack $n - 2t$ values together in a single $[\cdot]$-sharing by setting the degree of the underlying polynomials to $n - t - 1$. It is easy to note that robust reconstruction of such a $[\cdot]$-sharing is still possible, as there are $n - t$ honest parties in the set $\mathcal{P}$ and exactly $n - t$ shares are required to reconstruct an

$(n-t-1)$ degree polynomial. For every sub-circuit ckt_l, the w_{out_l} output values are grouped so that each group contains $n-2t$ secrets and each group is then converted to a single $[\cdot]$-sharing.

If we restrict to circuits for which any circuit wire has length at most $d/L = d/t$ (i.e. reaches upto at most d/L levels), then we ensure that the outputs of circuit ckt_l can only be the input to circuit ckt_{l+1}. With this restriction, the use of packed secret-sharing becomes applicable at all stages, and the communication complexity becomes $\mathcal{O}(|\mathsf{ckt}| \cdot (\frac{t}{d}+\kappa)\cdot\kappa^2)$, $\mathcal{BC}\big(|\mathsf{ckt}| \cdot \frac{t\cdot\kappa^2}{d}, n\big)$ and $\mathcal{BC}\big(|\mathsf{ckt}| \cdot \kappa^3, \kappa\big)$; i.e. a factor of n less in the first two terms compared to what is stated in Theorem 1. Realizing the broadcasts using DS and FH protocol respectively, we obtain the following corollaries:

Corollary 3. *If $d = \omega(\frac{n^3\cdot t}{\kappa^4})$ and if the calls to $\mathcal{F}_{\mathrm{BC}}$ are realized via the DS broadcast protocol, then the circuit-dependent communication complexity of Π_f is $\mathcal{O}(|\mathsf{ckt}| \cdot \kappa^7)$.*

Corollary 4. *If $d = \omega(\frac{n\cdot t}{\kappa^5})$ and $w = \omega(n^2\cdot(n+\kappa))$ (i.e. $|\mathsf{ckt}| = \omega(\frac{n^3\cdot t}{\kappa^5}(n+\kappa))$), then the circuit-dependent communication complexity of the protocol Π_f is $\mathcal{O}(|\mathsf{ckt}| \cdot \kappa^7)$.*

Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant EP/I03126X, and by Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079[10] and the third author was supported in part by a Royal Society Wolfson Merit Award.

References

1. M. Abe and S. Fehr. Adaptively Secure Feldman VSS and Applications to Universally-Composable Threshold Cryptography. In *Advances in Cryptology - CRYPTO 2004*, volume 3152 of *LNCS*, pages 317–334, 2004.
2. G. Asharov and Y. Lindell. A Full Proof of the BGW Protocol for Perfectly-Secure Multiparty Computation. *IACR Cryptology ePrint Archive*, 2011:136, 2011.
3. D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology - CRYPTO '91*, volume 576 of *LNCS*, pages 420–432, 1991.
4. Z. BeerliováTrubíniová and M. Hirt. Perfectly-Secure MPC with Linear Communication Complexity. In *Theory of Cryptography – TCC 2008*, volume 4948 of *LNCS*, pages 213–230, 2008.
5. E. Ben-Sasson, S. Fehr, and R. Ostrovsky. Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority. In *Advances in Cryptology - CRYPTO 2012*, volume 7417 of *LNCS*, pages 663–680, 2012.
6. E. Boyle, S. Goldwasser, and S. Tessaro. Communication locality in secure multi-party computation how to run sublinear algorithms in a distributed setting. In *Theory of Cryptography – TCC 2014*, volume 8349 of *LNCS*, pages 356–376, 2014.
7. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS*, pages 136–145, 2001.
8. R. Canetti and M. Fischlin. Universally Composable Commitments. In *Advances in Cryptology – CRYPTO 2001*, pages 19–40, 2001.
9. J. F. Canny and S. Sorkin. Practical large-scale distributed key generation. In *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 138–152, 2004.
10. A. Choudhury. Breaking the $\mathcal{O}(n|c|)$ barrier for unconditionally secure asynchronous multiparty computation - (extended abstract). In *Progress in Cryptology - INDOCRYPT 2013*, volume 8250 of *LNCS*, pages 19–37, 2013.
11. I. Damgård, Y. Ishai, M. Krøigaard, J.B. Nielsen, and A. Smith. Scalable Multiparty Computation with Nearly Optimal Work and Resilience. In *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *LNCS*, pages 241–261, 2008.
12. I. Damgård and C. Orlandi. Multiparty Computation for Dishonest Majority: From Passive to Active Security at Low Cost. In *Advances in Cryptology - CRYPTO 2010*, volume 6223 of *LNCS*, pages 558–576, 2010.

[10] The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

13. I. Damgård, V. Pastro, N.P. Smart, and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662, 2012.
14. V. Dani, V. King, M. Movahedi, and J. Saia. Brief Announcement: Breaking the $O(nm)$ Bit Barrier, Secure Multiparty Computation with a Static Adversary. In *Principles of Distributed Computing – PODC 2012*, pages 227–228, 2012.
15. V. Dani, V. King, M. Movahedi, and J. Saia. Quorums quicken queries: Efficient asynchronous secure multiparty computation. In *ICDN 2014*, volume 8314 of *LNCS*, pages 242–256, 2014.
16. D. Dolev and H. R. Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.
17. Uriel Feige. Noncryptographic selection protocols. In *FOCS*, pages 142–153, 1999.
18. M. Fitzi. *Generalized Communication and Security Models in Byzantine Agreement*. PhD thesis, ETH Zurich, 2002. `ftp://ftp.inf.ethz.ch/pub/crypto/publications/Fitzi03.pdf`.
19. Matthias Fitzi and Martin Hirt. Optimally Efficient Multi-valued Byzantine Agreement. In *Principles of Distributed Computing – PODC 2006*, pages 163–168. ACM, 2006.
20. M. K. Franklin and M. Yung. Communication Complexity of Secure Computation (Extended Abstract). In *Symposium on Theory of Computing – STOC 1992*, pages 699–710. ACM, 1992.
21. R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and Fact-Track Multiparty Computations with Applications to Threshold Cryptography. In *Principles of Distributed Computing – PODC '98*, pages 101–111. ACM, 1998.
22. O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.
23. B. M. Kapron, D. Kempe, V. King, J. Saia, and V. Sanwalani. Fast Asynchronous Byzantine Agreement and Leader Election with Full Information. *ACM Transactions on Algorithms*, 6(4), 2010.
24. V. King, S. Lonargan, J. Saia, and A. Trehan. Load Balanced Scalable Byzantine Agreement through Quorum Building, with Information. In *ICDCN*, volume 6522 of *LNCS*, pages 203–214, 2011.
25. V. King and J. Saia. Breaking the $O(n^2)$ Bit Barrier: Scalable Byzantine Agreement with an Adaptive Adversary. *J. ACM*, 58(4):18, 2011.
26. V. King, J. Saia, V. Sanwalani, and E. Vee. Scalable Leader Election. In *SODA*, pages 990–999, 2006.
27. T. P. Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Advances in Cryptology - CRYPTO '91*, volume 576 of *LNCS*, pages 129–140, 1992.
28. A. Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.
29. D. R. Stinson. *Cryptography - Theory and Practice*. Discrete mathematics and its applications series. CRC Press, 2005.

Appendices

A The UC Security Model

We work in the standard Universal Composability (UC) framework of Canetti [7], with static corruption. The UC framework introduces a PPT environment $\mathcal{Z}$ that is invoked on the computational security parameter κ, the statistical security parameter s and an auxiliary input z and oversees the execution of a protocol in one of the two worlds. The "ideal" world execution involves dummy parties $P_1, \ldots, P_n$, an ideal adversary $\mathcal{S}$ who may corrupt some of the dummy parties, and a functionality $\mathcal{F}$. The "real" world execution involves the PPT parties $P_1, \ldots, P_n$ and a real world adversary $\mathcal{A}$ who may corrupt some of the parties. In either of these two worlds, a PPT adversary can corrupt t parties out of the n parties. The environment $\mathcal{Z}$ chooses the input of the parties and may interact with the ideal world/real world adversary during the execution. At the end of the execution, it has to decide upon and output whether a real or an ideal world execution has taken place.

We let $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, s, z)$ denote the random variable describing the output of the environment $\mathcal{Z}$ after interacting with the ideal execution with adversary $\mathcal{S}$, the functionality $\mathcal{F}$, on the computational security parameter κ, the statistical security parameter s and z. Let $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote the ensemble $\{\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, s, z)\}_{\kappa,s\in\mathbb{N},z\in\{0,1\}^*}$. Similarly let $\text{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\kappa, s, z)$ denote the random variable describing the output of the environment $\mathcal{Z}$ after interacting in a real execution of a protocol Π with adversary $\mathcal{A}$, the parties $\mathcal{P}$, on the computational security parameter κ, the statistical security parameter s and z. Let $\text{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}$ denote the ensemble $\{\text{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\kappa, s, z)\}_{\kappa,s\in\mathbb{N},z\in\{0,1\}^*}$.

Definition 4. *For $n \in \mathbb{N}$, let $\mathcal{F}$ be an n-ary functionality and let Π be an n-party protocol. We say that Π (κ, s)-securely realizes $\mathcal{F}$ in the UC security framework, if for every PPT real world adversary $\mathcal{A}$, there exists a PPT ideal world adversary $\mathcal{S}$, corrupting the same parties, such that the following two distributions are computationally indistinguishable in κ, with all but 2^{-s} probability:*

$$\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \overset{c}{\approx} \text{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}.$$

We consider the above definition where it quantifies over different adversaries: passive or active, that corrupts only certain number of parties. Note that the security offered by the statistical security parameter s does not depend upon the computational power of the adversary.

Modular Composition: A great advantage of the UC model is that it allows to prove the security of the protocols in a modular fashion. Specifically, the sequential modular composition theorem [7] states that in order to analyze the security of a protocol π_f for computing a function f that uses a subprotocol π_g for computing another function g, it suffices to consider the execution of π_f in a model where a trusted third party is used to ideally compute g (instead of the parties running the real subprotocol π_g). This facilitates a modular analysis of security: we first prove the security of π_g (as per the UC definition) and then prove the security of π_f assuming an ideal party (functionality) for g. This model in which π_f is analyzed using ideal calls to g, instead of executing π_g, is called the *g-hybrid model* because it involves both a real protocol execution (for computing f) and an ideal trusted third party computing g.

B UC-secure Instantiation of Various Functionalities

B.1 Protocol for Realizing $\mathcal{F}_{\text{GEN}[\cdot]}$

We design a protocol $\Pi_{[\cdot]}$, presented in Figure 7, for realizing the functionality $\mathcal{F}_{\text{GEN}[\cdot]}$ in the UC framework. We closely follow the standard Pedersen VSS scheme [27] against a threshold static adversary. Specifically, let $\mathbf{C}$ be the existing commitment available to the parties in $\mathcal{P}$ such that $\mathbf{C} = \mathsf{Comm}_{\mathsf{ck}}(s; g, h)$ and let (s, g, h) be available to D. To $[\cdot]$-share s, the dealer D selects three random polynomials $f(\cdot), g(\cdot)$ and $h(\cdot)$ each of degree at most t such that $f(0) = s, g(0) = g$ and $h(0) = h$. To every party P_i in $\mathcal{P}$, D distributes the share $f_i = f(\alpha_i)$ and opening information $g_i = g(\alpha_i)$ and $h_i = h(\alpha_i)$. Additionally, D publicly commits to the shares of all the share-holders, with the corresponding opening information acting as the randomness for the commitments. Namely D broadcasts $\{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}}$ via $\mathcal{F}_{\text{BC}}$.

Every honest party P_i then verifies three conditions: **(1).** if the commitments correspond to polynomials of degree at most t **(2).** if the commitments are consistent with $\mathbf{C}$ in the sense that the constant terms of the polynomials committed via the commitments $\{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}}$ are indeed embedded in $\mathbf{C}$ **(3).** if f_i, g_i, h_i received over the point-to-point channel is consistent with $\mathbf{C}_{f_i,g_i,h_i}$ received via $\mathcal{F}_{\text{BC}}$. The first two tests can be done appealing to the homomorphic property of the commitment scheme. If any of the first two tests fails, then P_i concludes that D is corrupted and outputs $\mathsf{Failure}$. If the last test fails (but first two tests succeed), then P_i complains D (publicly) who resolves the complain by revealing f_i, g_i, h_i via $\mathcal{F}_{\text{BC}}$. Subsequently, the third test is checked with f_i, g_i, h_i received from D publicly. If the test is successful, P_i accepts the new f_i, g_i, h_i and outputs $[s]_i = (f_i, g_i, h_i, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}})$. Else P_i outputs $\mathsf{Failure}$.

Intuitively the privacy of the shared secret s for an honest D follows from the fact that $\mathcal{A}$ may learn at most t shares, which constitute t distinct points on $f(\cdot)$ having degree t; so from adversary's point of view, we have one "degree of freedom"; i.e. for every possible choice of s, there exists a unique $f(\cdot)$ polynomial of degree t, which is consistent with the shares received by $\mathcal{A}$. Note that the publicly known commitment

of the shares do not provide any additional information about the unknown shares to $\mathcal{A}$, thanks to the (statistical) hiding property of the commitment scheme and the fact that the corresponding randomness lie on polynomials of degree at most t and $\mathcal{A}$ will be provided with at most t points on them, again implying one degree of freedom.

Protocol $\Pi_{[\cdot]}$

The public input to the protocol is a publicly known commitment $\mathbf{C}$ available to the parties in $\mathcal{P}$, while the private input for D is a secret s and randomness pair (g, h), such that $\mathbf{C} = \mathsf{Comm_{ck}}(s; g, h)$ holds. For session ID sid, D and the parties in $\mathcal{P}$ do the following:

Round 1 (Share Distribution and Broadcasting Commitments) — D does the following:
- Select three random polynomials $f(\cdot), g(\cdot)$ and $h(\cdot)$ of degree at most t, subject to the condition that $f(0) = s, g(0) = g$ and $h(0) = h$.
- Corresponding to every $P_i \in \mathcal{P}$, compute the *share* $f_i = f(\alpha_i)$ and the *opening information* $g_i = g(\alpha_i), h_i = h(\alpha_i)$ and the commitment $\mathbf{C}_{f_i,g_i,h_i} = \mathsf{Comm_{ck}}(f_i; g_i, h_i)$.
- Corresponding to every $P_i \in \mathcal{P}$, send $(\mathsf{sid}, i, (f_i, g_i, h_i))$ to the party P_i. In addition, call $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, \mathsf{D}, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}}, \mathcal{P})$.

Round 2 (Consistency Verification and Complaints) — Every party $P_i \in \mathcal{P}$ does the following:
- Receive $(\mathsf{sid}, i, \mathsf{D}, (f_i, g_i, h_i))$ from D and $(\mathsf{sid}, i, \mathsf{D}, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}})$ from $\mathcal{F}_{\text{BC}}$.
- Using the homomorphic property of commitments, verify
  - if there exists polynomials of degree at most t, say $f'(\cdot), g'(\cdot)$ and $h'(\cdot)$ such that $\mathbf{C}_{f_j,g_j,h_j}$ is $\mathsf{Comm_{ck}}(f'(\alpha_j); g'(\alpha_j), h'(\alpha_j))$ [a] for every $P_j \in \mathcal{P}$.
  - whether the $\mathbf{C}$ is same as $\mathsf{Comm_{ck}}(f'(0); g'(0), h'(0))$.

  If any of the above tests fail then output $(\mathsf{sid}, i, \mathsf{Failure})$ and halt.
- Verify whether $\mathbf{C}_{f_i,g_i,h_i} \stackrel{?}{=} \mathsf{Comm_{ck}}(f_i; g_i, h_i)$. If the verification fails then call $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, i, (\texttt{Unhappy}, \mathsf{D}), \mathcal{P})$.

Local Computation (at the end of Round 2) — Every party P_i in $\mathcal{P}$ does the following:
- Construct a set $\mathcal{W}_i$ initialized to $\emptyset$ and add $P_j \in \mathcal{P}$ to $\mathcal{W}_i$ if corresponding to party P_j the message $(\mathsf{sid}, i, j, (\texttt{Unhappy}, \mathsf{D}))$ is received from[b] $\mathcal{F}_{\text{BC}}$.
- If $|\mathcal{W}_i| > t$, then output $(\mathsf{sid}, i, \mathsf{Failure})$ and halt.

Round 3 (Resolving Complaints) — D does the following:
- Corresponding to each $P_i \in \mathcal{W}_{\mathsf{D}}$, call $\mathcal{F}_{\text{BC}}$ with the message $(\mathsf{sid}, \mathsf{D}, (\texttt{Resolve}, P_i, f_i, g_i, h_i), \mathcal{P})$.

Local Computation (at the end of Round 3) — every party $P_i \in \mathcal{P}$ does the following:
- If there exists a $P_k \in \mathcal{W}_i$ corresponding to which the message $(\mathsf{sid}, i, \mathsf{D}, (\texttt{Resolve}, P_k, f_k, g_k, h_k))$ is received from $\mathcal{F}_{\text{BC}}$ such that $\mathbf{C}_{f_k,g_k,h_k} \neq \mathsf{Comm_{ck}}(f_k; g_k, h_k)$, then output $(\mathsf{sid}, i, \mathsf{Failure})$ and halt.
- Else output $[s]_i$ computed as follows and halt:
  - If $P_i \in \mathcal{P} \setminus \mathcal{W}_i$, then $[s]_i = (f_i, g_i, h_i, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}})$ where f_i, g_i, h_i is received from D in **Round 1**.
  - If $P_i \in \mathcal{W}_i$, then $[s]_i = (f_i, g_i, h_i, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}})$ where f_i, g_i, h_i is received from D in **Round 3**.

[a] This is done using a standard procedure based on the properties of Vandermonde matrix; see for example [21].
[b] The contents of $\mathcal{W}_i$ will be the same for each honest party P_i in $\mathcal{P}$.

Fig. 7. Protocol for UC-secure realizing $\mathcal{F}_{\text{GEN}[\cdot]}$

The properties of the protocol $\Pi_{[\cdot]}$ are formally stated in Lemma 5.

Lemma 5. *Let $\mathsf{D} \in \mathcal{P}$ be a dealer with secret s and randomness pair (g, h) and let $\mathbf{C}$ be a publicly known commitment available to the parties in $\mathcal{P}$. Then the protocol $\Pi_{[\cdot]}$ UC-securely realizes the functionality $\mathcal{F}_{\text{GEN}[\cdot]}$ in the $\mathcal{F}_{\text{BC}}$-hybrid model. The protocol has communication complexity $\mathcal{O}(n\kappa)$ bits and $\mathcal{BC}(n\kappa, n)$.*

PROOF: The communication complexity follows easily from the protocol. We next prove the security, considering the following two cases.

Case I — When D is honest: We first claim that in this case, no honest party will output $\mathsf{Failure}$; this easily follows from the fact that an honest D will distribute consistent shares and only the corrupted share-

holders (at most t) will accuse D and such accusations will be resolved correctly by D. Let $T \subset \mathcal{P}$ be the set of parties under the control of $\mathcal{A}$ during the protocol $\Pi_{[\cdot]}$; we present a simulator $\mathcal{S}^{\text{HD}}_{[\cdot]}$ (interacting with the functionality $\mathcal{F}_{\text{GEN}[\cdot]}$) for $\mathcal{A}$ in Figure 8. The high level idea behind the simulator is the following: the simulator interacts with $\mathcal{F}_{\text{GEN}[\cdot]}$ and obtains the shares and opening information of the corrupted parties, along with all the committed shares and sends the same to the real-world adversary; the simulator then simulates the rest of the protocol steps of $\Pi_{[\cdot]}$ on the behalf of the honest parties (including D). Any accusation by a (corrupted) share-holder can be easily resolved by the simulator, as it knows the corresponding share and opening information (as obtained from the functionality), which it can reveal. It follows easily that the simulated view has exactly the same distribution as the view of the real-world adversary in $\Pi_{[\cdot]}$.

Simulator $\mathcal{S}^{\text{HD}}_{[\cdot]}$

The simulator plays the role of the honest parties (including D) and simulates each step of the protocol $\Pi_{[\cdot]}$ as follows. The communication of the $\mathcal{Z}$ with the adversary $\mathcal{A}$ is handled as follows: Every input value received by the simulator from $\mathcal{Z}$ is written on $\mathcal{A}$'s input tape. Likewise, every output value written by $\mathcal{A}$ on its output tape is copied to the simulator's output tape (to be read by the environment $\mathcal{Z}$). The simulator then does the following for the session ID sid:

- Interact with $\mathcal{F}_{\text{GEN}[\cdot]}$ and obtain $(\mathsf{sid}, i, (f_i, g_i, h_i, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}}))$ corresponding to every corrupted party $P_i \in T$.
- On the behalf of D, send $(\mathsf{sid}, i, (f_i, g_i, h_i))$ to $\mathcal{A}$, corresponding to every $P_i \in T$. In addition, send $(\mathsf{sid}, \mathsf{D}, i, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}})$ to $\mathcal{A}$ on the behalf of $\mathcal{F}_{\text{BC}}$, corresponding to each $P_i \in T$.
- On receiving $(\mathsf{sid}, i, \mathcal{P}, (\texttt{Unhappy}, \mathsf{D}))$ as the message to $\mathcal{F}_{\text{BC}}$ from $\mathcal{A}$ on the behalf of any $P_i \in T$, send $(\mathsf{sid}, \mathsf{D}, i, (\texttt{Resolve}, P_i, f_i, g_i, h_i))$ to $\mathcal{A}$ as the message from $\mathcal{F}_{\text{BC}}$ on the behalf of D.

The simulator then outputs $\mathcal{A}$'s output and terminate.

Fig. 8. Simulator for the adversary $\mathcal{A}$ corrupting at most t parties in the set $T \subset \mathcal{P} \setminus \mathsf{D}$ in the protocol $\Pi_{[\cdot]}$.

Case 2 — When D is Corrupted: We first note that there exists at least $t+1$ honest parties in $\mathcal{P}$ and that there exists only a unique polynomial of degree at most t passing through a set of $t+1$ or more distinct points. With these facts, we next prove the security with respect to a corrupted D. Let $T \subset \mathcal{P}$ be the set of parties under the control of $\mathcal{A}$ including D, during the protocol $\Pi_{[\cdot]}$; we present a simulator $\mathcal{S}^{\text{CD}}_{[\cdot]}$ (interacting with the functionality $\mathcal{F}_{\text{GEN}[\cdot]}$) for $\mathcal{A}$ in Figure 9.

It follows easily that the simulated view is computationally indistinguishable from the view of the real-world adversary; otherwise we can use the corresponding distinguisher to break the binding property of the underlying commitment scheme. □

Simulator $\mathcal{S}^{\mathbf{CD}}_{[\cdot]}$

The simulator plays the role of the honest parties and simulates each step of the protocol $\Pi_{[\cdot]}$ as follows. The communication of the $\mathcal{Z}$ with the adversary $\mathcal{A}$ is handled as follows: Every input value received by the simulator from $\mathcal{Z}$ is written on $\mathcal{A}$'s input tape. Likewise, every output value written by $\mathcal{A}$ on its output tape is copied to the simulator's output tape (to be read by the environment $\mathcal{Z}$). The simulator then does the following for the session ID sid:

- Play the role of $n - |T|$ honest parties and interact with $\mathcal{A}$ as per the protocol $\Pi_{[\cdot]}$.
- If Failure is obtained during the simulated execution of the protocol due to the fact that the committed shares and the corresponding randomness do not lie on polynomials of degree at most t, then send three arbitrary polynomials of degree more than t on the behalf of D to the functionality $\mathcal{F}_{\text{GEN}[\cdot]}$.
- Else define three polynomials $\hat{f}(\cdot), \hat{g}(\cdot)$ and $\hat{h}(\cdot)$ of degree at most t, such that $\hat{f}(\alpha_i) = f_i, \hat{g}(\alpha_i) = g_i$ and $\hat{h}(\alpha_i) = h_i$ holds for every honest party $P_i \notin T$, where f_i and (g_i, h_i) are the corresponding share and opening information respectively which are obtained by P_i during the simulated run of [a] of $\Pi_{[\cdot]}$. Then send the polynomials $\hat{f}(\cdot), \hat{g}(\cdot)$ and $\hat{h}(\cdot)$ on the behalf of D to $\mathcal{F}_{\text{GEN}[\cdot]}$.

The simulator then outputs $\mathcal{A}$'s output and terminate.

[a] Note that $\hat{f}(\cdot), \hat{g}(\cdot)$ and $\hat{h}(\cdot)$ are well defined as there exists $|n| - |T| > t + 1$ honest parties in $\mathcal{P}$.

Fig. 9. Simulator for the adversary $\mathcal{A}$ corrupting at most t parties in the set $T \subset \mathcal{P}$ including D during the protocol $\Pi_{[\cdot]}$.

B.2 Protocols for Realizing $\mathcal{F}_{\text{COMMITTEE}}$ and $\mathcal{F}_{\text{BC}}$

The Committee Selection Protocol: Functionality $\mathcal{F}_{\text{COMMITTEE}}$ can be realized using various standard ways; moreover, the functionality will be invoked at most $(t + 1)$ times in our MPC protocol; t times corresponding to t failed sub-circuit evaluations plus once for initial selection of a committee. As this cost is independent of the circuit size $|\mathsf{ckt}|$ (but rather $\mathsf{Poly}(n)$), we give only a high level sketch of one of the possible instantiations of $\mathcal{F}_{\text{COMMITTEE}}$, based on a computationally secure pseudo-random number generator (PRNG) [29]. Assume we have a PRNG $\mathcal{R}_k(\cdot)$ with seed k, which outputs values in the range $1, \ldots, n$. Then each time a committee needs to be formed, the parties in $\mathcal{P}$ can agree on a random seed k; this can be done via standard method, say by coin-flipping (or executing an instance of $\Pi_{[\cdot]}$ on the behalf of each party). Then the parties can (locally) run $\mathcal{R}$ with the obtained key, till they obtain the desired committee. It follows via the security of $\mathcal{R}$, that the committee selected like this is indeed a uniformly random committee of parties with high probability. We can simplify further by putting up a random seed in the CRS, rather than sampling a random seed on the fly every time a committee needs to be formed.

The Broadcast Protocol: Assuming a PKI set-up, the well known Dolev-Strong (DS) broadcast protocol [16] allows a sender $\mathsf{Sen} \in \mathcal{P}$ to reliably broadcast a message m of size ℓ to a set of parties $\mathcal{X} \subseteq \mathcal{P}$, provided $\mathcal{X}$ has at least one honest party; the protocol can be used to realize $\mathcal{F}_{\text{BC}}$. As stated in [19], using the DS protocol, it costs the parties in $\mathcal{X} \cup \{\mathsf{Sen}\}$ a total communication of $\mathcal{O}(|\mathcal{X}|^3 \cdot \ell \cdot \kappa)$ bits over the point-to-point channels to enable the Sen to broadcast m to the parties in $\mathcal{X}$. As the protocol is well known in the literature, we avoid giving the details here and instead refer the interested readers to [18] for the details. We also note that [19] suggests an improved proposal for realizing $\mathcal{F}_{\text{BC}}$ with a communication complexity of $\mathcal{O}(|\mathcal{X}| \cdot \ell + |\mathcal{X}|^4 \cdot (|\mathcal{X}| + \kappa))$ bits, but with a restriction on the size of ℓ, namely $\ell = \omega(|\mathcal{X}|^2 \cdot (|\mathcal{X}| + \kappa))$. We make use of this proposal for estimating the communication complexity of our MPC protocol in Section 3.3.

C Supporting Sub-Protocols

In this appendix, we present the details for the sub-protocols which enable a number of tasks such as conversion from $[\cdot]$-sharing to $\langle\cdot\rangle$-sharing and vice-versa and generating a random $[\cdot]$-sharing of 0.

C.1 Protocol for Converting a $\langle\cdot\rangle$-sharing to a $[\cdot]$-sharing

Protocol $\Pi_{\langle\cdot\rangle\to[\cdot]}$ is presented in Figure 10.

Protocol $\Pi_{\langle\cdot\rangle\to[\cdot]}$

For session id sid, every party $P_i \in \mathcal{P}$ participates with either $(\mathsf{sid}, i, \langle s\rangle_i)$ or (sid, i) and does the following:

- If $P_i \in \mathcal{X}$, interpret $\langle s\rangle_i$ as $(s_i, u_i, v_i, \{\mathbf{C}^{P_i}_{s_j,u_j,v_j}\}_{P_j\in\mathcal{X}})$ and invoke $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, i, \{\mathbf{C}^{P_i}_{s_j,u_j,v_j}\}_{P_j\in\mathcal{X}}, \mathcal{P})$
- Receive $(\mathsf{sid}, i, k, \{\mathbf{C}^{P_k}_{s_j,u_j,v_j}\}_{P_j\in\mathcal{X}})$ from $\mathcal{F}_{\text{BC}}$ for every $P_k \in \mathcal{X}$ (who acted as the sender).
- If there exists a pair of parties $P_a, P_b \in \mathcal{X}$, such that $\{\mathbf{C}^{P_a}_{s_j,u_j,v_j}\}_{P_j\in\mathcal{X}} \neq \{\mathbf{C}^{P_b}_{s_j,u_j,v_j}\}_{P_j\in\mathcal{X}}$, then output $(\mathsf{sid}, i, \mathsf{Failure}, P_a, P_b)$ and halt; if there are multiple such pairs (P_a, P_b) the select the one with the least index a and b. Else set $\{\mathbf{C}_{s_j,u_j,v_j}\}_{P_j\in\mathcal{X}} = \{\mathbf{C}^{P_\alpha}_{s_j,u_j,v_j}\}_{P_j\in\mathcal{X}}$ to be the reference set of commitments, where P_α is the least indexed party in $\mathcal{P}$.
- If $P_i \in \mathcal{X}$, act as a D and call $\mathcal{F}_{\text{GEN}[\cdot]}$ with $(\mathsf{sid}, i, f^{(i)}(\cdot), g^{(i)}(\cdot), h^{(i)}(\cdot))$ where $f^{(i)}(\cdot), g^{(i)}(\cdot)$ and $h^{(i)}(\cdot)$ are random polynomials of degree at most t, subject to the condition that $f^{(i)}(0) = s_i, g^{(i)}(0) = u_i$ and $h^{(i)}(0) = v_i$. If $P_i \in \mathcal{P}\setminus\mathcal{X}$, invoke $\mathcal{F}_{\text{GEN}[\cdot]}$ with $(\mathsf{sid}, i, k, \mathbf{C}_{s_k,u_k,v_k})$ for every $P_k \in \mathcal{X}$, where $\mathbf{C}_{s_k,u_k,v_k}$ is obtained from the reference set of commitments. Receive $(\mathsf{sid}, i, k, \mathsf{Failure})$ or $(\mathsf{sid}, i, k, [s_k]_i)$ from $\mathcal{F}_{\text{GEN}[\cdot]}$ for every $P_k \in \mathcal{X}$
- Output $(\mathsf{sid}, i, \mathsf{Failure}, P_k)$ and halt if $(\mathsf{sid}, i, P_k, \mathsf{Failure})$ is received from $\mathcal{F}_{\text{GEN}[\cdot]}$ corresponding to any $P_k \in \mathcal{X}$. Otherwise, locally compute $[s]_i = \sum_{P_k\in\mathcal{X}}[s_k]_i$, output $(\mathsf{sid}, i, \mathsf{Success}, [s]_i)$ and halt.

Fig. 10. Protocol for Converting $\langle\cdot\rangle$-sharing to $[\cdot]$-sharing in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{GEN}[\cdot]})$-hybrid Model

C.2 Protocol for Generating $\langle\cdot\rangle$-sharing of a Committed Secret

Protocol $\Pi_{\langle\cdot\rangle}$ is presented in Figure 11.

Protocol $\Pi_{\langle\cdot\rangle}$

For session ID sid, every $P_i \in \mathcal{P}$ participates with $(\mathsf{sid}, i, \mathsf{D}, \mathbf{C}_{f,g,h}, \mathcal{X})$ where $\mathbf{C}_{f,g,h}$ is a (publicly known) commitment. The dealer D participates with $(\mathsf{sid}, \mathsf{D}, f, g, h, \mathcal{X})$ where f is the secret and (g, h) is the randomness pair, such that $\mathbf{C} = \mathsf{Comm}_{\mathsf{ck}}(f; g, h)$. The parties in $\mathcal{P}$ do the following:

Round 1 (Share Distribution and Broadcasting Commitments): Only D does the following:
- Corresponding to every $P_i \in \mathcal{X}$, select a random *share* s_i and a random pair of *opening information* u_i, v_i, subject to the condition that $\sum_{P_i \in \mathcal{X}} s_i = f, \sum_{P_i \in \mathcal{X}} u_i = g$ and $\sum_{P_i \in \mathcal{X}} v_i = h$, and compute the commitment $\mathbf{C}_{s_i,u_i,v_i} = \mathsf{Comm}_{\mathsf{ck}}(s_i; u_i, v_i)$. Send $(\mathsf{sid}, i, \mathsf{D}, s_i, u_i, v_i)$ to the party P_i.
- Call $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, \mathsf{D}, \{\mathbf{C}_{s_j,u_j,v_j}\}_{P_j \in \mathcal{X}}, \mathcal{P})$ to broadcast $\{\mathbf{C}_{s_j,u_j,v_j}\}_{P_j \in \mathcal{X}}$ to all the parties in $\mathcal{P}$.

Round 2 (Consistency Verification and Complaints): Every party $P_i \in \mathcal{P}$ does the following:
- Receive $(\mathsf{sid}, i, \mathsf{D}, \{\mathbf{C}_{s_j,u_j,v_j}\}_{P_j \in \mathcal{X}})$ from $\mathcal{F}_{\text{BC}}$. Additionally if $P_i \in \mathcal{X}$, then receive $(\mathsf{sid}, i, \mathsf{D}, s_i, u_i, v_i)$ from D.
- Verify if $\odot_{P_j \in \mathcal{X}} \mathbf{C}_{s_j,u_j,v_j} \stackrel{?}{=} \mathbf{C}_{f,g,h}$ (homomorphically). If the verification fails, then output $(\mathsf{sid}, i, \mathsf{D}, \mathsf{Failure})$ and halt.
- If $P_i \in \mathcal{X}$ then verify whether $\mathbf{C}_{s_i,u_i,v_i} \stackrel{?}{=} \mathsf{Comm}_{\mathsf{ck}}(s_i; u_i, v_i)$. If the verification fails then call $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, i, (\texttt{Unhappy}, i, \mathsf{D}), \mathcal{P})$.
- Construct a set $\mathcal{W}_i$ initialized to $\emptyset$ and add $P_j \in \mathcal{X}$ to $\mathcal{W}_i$ if $(\mathsf{sid}, i, j, (\texttt{Unhappy}, j, \mathsf{D}))$ is received from[a] $\mathcal{F}_{\text{BC}}$ corresponding to P_j.

Round 3 (Resolving Complaints): Only D does the following:
- Corresponding to each $P_i \in \mathcal{W}_{\mathsf{D}}$, call $\mathcal{F}_{\text{BC}}$ with the message $(\mathsf{sid}, \mathsf{D}, (\texttt{Resolve}, i, s_i, u_i, v_i), \mathcal{P})$.

Local Computation (at the end of Round 3): Every party $P_i \in \mathcal{P}$ does the following:
- If there exists a $P_k \in \mathcal{W}_i$ corresponding to which the message $(\mathsf{sid}, i, \mathsf{D}, (\texttt{Resolve}, k, s_k, u_k, v_k))$ is received from $\mathcal{F}_{\text{BC}}$ such that $\mathbf{C}_{s_k,u_k,v_k} \neq \mathsf{Comm}_{\mathsf{ck}}(s_k; u_k, v_k)$, then output $(\mathsf{sid}, i, \mathsf{D}, \mathsf{Failure})$ and halt.
- Else every $P_i \in \mathcal{P} \setminus \mathcal{X}$ outputs $(\mathsf{sid}, i, \mathsf{D}, \mathsf{Success})$ and halts, while every $P_i \in \mathcal{X}$ does the following:
  - If $P_i \in \mathcal{X} \setminus \mathcal{W}_i$, then set $\langle f \rangle_i = (s_i, u_i, v_i, \{\mathbf{C}_{s_j,u_j,v_j}\}_{P_j \in \mathcal{X}})$, where (s_i, u_i, v_i) was received from D and $\{\mathbf{C}_{s_j,u_j,v_j}\}_{P_j \in \mathcal{X}}$ was received from $\mathcal{F}_{\text{BC}}$ at the end of **Round 1**. Output $(\mathsf{sid}, i, \mathsf{D}, \mathsf{Success}, \langle f \rangle_i)$ and halt.
  - Else if $P_i \in \mathcal{W}_i$, then set $\langle f \rangle_i = (s_i, u_i, v_i, \{\mathbf{C}_{s_j,u_j,v_j}\}_{P_j \in \mathcal{X}})$, where (s_i, u_i, v_i) was received from $\mathcal{F}_{\text{BC}}$ (corresponding to D) at the end of **Round 3** and $\{\mathbf{C}_{s_j,u_j,v_j}\}_{P_j \in \mathcal{X}}$ was received from $\mathcal{F}_{\text{BC}}$ at the end of **Round 1**. Output $(\mathsf{sid}, i, \mathsf{Success}, \mathsf{D}, \langle f \rangle_i)$ and halt.

[a] The contents of $\mathcal{W}_i$ will be the same for each honest party P_i in $\mathcal{P}$.

Fig. 11. Protocol $\Pi_{\langle\cdot\rangle}$ for Verifiably $\langle\cdot\rangle$-sharing an Existing Committed Secret

C.3 Protocol for Transforming $[\cdot]$-sharing to $\langle\cdot\rangle$-sharing

Protocol $\Pi_{[\cdot]\to\langle\cdot\rangle}$ is presented in Figure 12.

Protocol $\Pi_{[\cdot]\to\langle\cdot\rangle}$

For session ID sid, every party $P_i \in \mathcal{P}$ participates in the protocol with $(\mathsf{sid}, i, [s]_i, \mathcal{X})$, where $[s]_i = (f_i, g_i, h_i, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j\in\mathcal{P}})$ and does the following:

Verifiably $\langle\cdot\rangle$-sharing the Share and Opening Information in $[s]_i$. Act as a dealer D and participate in an instance of $\Pi_{\langle\cdot\rangle}$ with input $(\mathsf{sid}, i, f_i, g_i, h_i, \mathcal{X})$. For every $P_k \in \mathcal{P}$, participate in the instance of $\Pi_{\langle\cdot\rangle}$ corresponding to the dealer P_k with input $(\mathsf{sid}, i, k, \mathbf{C}_{f_k,g_k,h_k}, \mathcal{X})$.

Identifying the Correctly $\langle\cdot\rangle$-shared Shares of s and Generating $\langle s\rangle_{\mathcal{X}}$. If $P_i \in \mathcal{P}\setminus\mathcal{X}$, then output (sid, i) and halt. Else construct a set $\mathcal{H}$ initialized to $\emptyset$.

- Include $P_k \in \mathcal{P}$ to $\mathcal{H}$ if $(\mathsf{sid}, i, k, \mathsf{Success}, \langle f_k\rangle_i)$ is the output for the instance of $\Pi_{\langle\cdot\rangle}$ where P_k acted as the dealer.
- Without loss of generality, let[a] $\mathcal{H} = \{P_1, \ldots, P_{|\mathcal{H}|}\}$ and let $c_1, \ldots, c_{|\mathcal{H}|}$ be the publicly known Lagrange interpolation coefficients, such that $c_1 f_1 + \ldots + c_{|\mathcal{H}|} f_{|\mathcal{H}|} = s$. Then locally compute $\langle s\rangle_i = c_1\langle f_1\rangle_i + \ldots + c_{|\mathcal{H}|}\langle f_{|\mathcal{H}|}\rangle_i$, output $(\mathsf{sid}, i, \langle s\rangle_i)$ and halt.

[a] The set $\mathcal{H}$ will be of size more than $t+1$.

Fig. 12. Protocol $\Pi_{[\cdot]\to\langle\cdot\rangle}$ for Converting an $[\cdot]$-sharing to $\langle\cdot\rangle$-sharing.

C.4 Protocol for Generating Random $[\cdot]$-sharing of 0

As mentioned earlier, the protocol uses the ideal ZK functionality $\mathcal{F}_{\text{ZK.BC}}$ presented in Figure 13.

Functionality $\mathcal{F}_{\text{ZK.BC}}$

$\mathcal{F}_{\text{ZK.BC}}$ interacts with a prover $P_j \in \mathcal{P}$ and the set of n verifiers $\mathcal{P} = \{P_1, \ldots, P_n\}$ and the adversary $\mathcal{S}$ and is parameterized by the commitment key ck of a double-trapdoor homomorphic commitment scheme.

- Upon receiving $(\mathsf{sid}, j, \mathbf{C}, u, v)$ from the prover P_j and (sid, j, i) from every party $P_i \in \mathcal{P}\setminus\{P_j\}$, the functionality sends $(\mathsf{sid}, i, \mathbf{C})$ to every party $P_i \in \mathcal{P}$ and $\mathcal{S}$ and halts if $\mathbf{C} \stackrel{?}{=} \mathsf{Comm}_{\mathsf{ck}}(0; u, v)$ is true. Else the functionality sends $(\mathsf{sid}, i, \perp)$ to every party $P_i \in \mathcal{P}$ and $\mathcal{S}$ and halts.

Fig. 13. The Ideal Functionality for ZK Proof of Committing Zero

Now based on the functionalities $\mathcal{F}_{\text{GEN}[\cdot]}$ and $\mathcal{F}_{\text{ZK.BC}}$, protocol $\Pi_{\text{RANDZERO}[\cdot]}$ is presented in Figure 14.

Protocol $\Pi_{\text{RANDZERO}[\cdot]}$

For the session id sid, every party $P_i \in \mathcal{P}$ participates with (sid, i) and does the following:

Publicly Committing 0:
- Set $r_i = 0$ and randomly select $u_i, v_i \in \mathbb{F}_p$ and compute $\mathbf{C}_{r_i,u_i,v_i} = \mathsf{Comm}_{\mathsf{ck}}(r_i; u_i, v_i)$.
- Act as a prover and call $\mathcal{F}_{\text{ZK.BC}}$ with $(\mathsf{sid}, i, \mathbf{C}_{r_i,u_i,v_i}, u_i, v_i)$. Corresponding to every prover $P_j \in \mathcal{P} \setminus P_i$, participate in $\mathcal{F}_{\text{ZK.BC}}$ with (sid, j, i).
- Construct a set $\mathcal{T}_i$, initialized to $\emptyset$ and include P_j in $\mathcal{T}_i$ if corresponding to the prover P_j, $(\mathsf{sid}, i, \mathbf{C}_{r_i,u_j,v_j})$ is received from $\mathcal{F}_{\text{ZK.BC}}$.

$[\cdot]$**-sharing** 0:
- Select three random polynomials $f^{(i)}(\cdot), g^{(i)}(\cdot)$ and $h^{(i)}(\cdot)$ each of degree at most t, subject to the condition that $f^{(i)}(0) = r_i, g^{(i)}(0) = u_i$ and $h^{(i)}(0) = v_i$.
- Act as a D and call $\mathcal{F}_{\text{GEN}[\cdot]}$ with $(\mathsf{sid}, P_i, f^{(i)}(\cdot), g^{(i)}(\cdot), h^{(i)}(\cdot))$. Corresponding to every dealer $P_j \in \mathcal{T}_i$, participate in $\mathcal{F}_{\text{GEN}[\cdot]}$ with $(\mathsf{sid}, i, j, \mathbf{C}_{r_j,u_j,v_j})$.
- If corresponding to any $P_j \in \mathcal{T}_i$, $(\mathsf{sid}, i, P_j, \mathsf{Failure})$ is received from $\mathcal{F}_{\text{GEN}[\cdot]}$, then remove P_j from $\mathcal{T}_i$.
- Locally compute $[0]_i = \sum_{P_j \in \mathcal{T}_i} [r_j]_i$, where $(\mathsf{sid}, i, j, [r_j]_i)$ is received from $\mathcal{F}_{\text{GEN}[\cdot]}$ corresponding to $P_j \in \mathcal{T}_i$. Output $(\mathsf{sid}, i, [0]_i)$ and halt.

Fig. 14. Protocol $\Pi_{\text{RANDZERO}[\cdot]}$ for generating a random $[\cdot]$-sharing of 0.

D Protocol $\Pi_{\mathsf{C}}^{\text{NR}}$ for $\langle\cdot\rangle$-shared Evaluation of a Circuit

As evident from the high level description of protocol $\Pi_{\mathsf{C}}^{\text{NR}}$ in Section 3.2, the major step of the protocol $\Pi_{\mathsf{C}}^{\text{NR}}$ is the preparation stage for generating the shared-triplets. Towards constructing the preparation stage protocol and protocol $\Pi_{\mathsf{C}}^{\text{NR}}$, we begin with the building blocks and sub-protocols most of which are taken from [12] and rest are modified according to our need. Many of the sub-protocols are described with respect to a set of parties $\mathcal{X} \subset \mathcal{P}$, where we assume that $\mathcal{X}$ contains at least one honest party.

Strong Semi-honest Secure Two-party Multiplication Protocol. Protocol $\Pi_{\text{MULT}}(a, b) \rightarrow (c_1, c_2)$ is a two-party protocol. The inputs of the first and second party are a and b respectively. The outputs to the first and second party are c_1 and c_2 respectively. It holds that c_1 is random in $\mathbb{F}_p$ and $c_1 + c_2 = a \cdot b$. Informally the protocol satisfies the following properties (for the complete formal details see [12]):

- The protocol is secure even if the adversary maliciously chooses the randomness for the corrupted parties (this is the reason [12] calls the protocol as *strong semi-honest* secure).
- The view of the protocol commits the adversary to his randomness and given the view and the randomness it is possible to verify whether any party deviated from the protocol.

In our context, the second property of Π_{MULT} is very crucial, as it enables an honest party involved in Π_{MULT} to identify any malicious behavior of its partner in the protocol when the individual randomness are revealed. There are various standard ways for instantiating $\Pi_{\text{MULT}}(a, b)$, based on variety of standard assumptions, such as homomorphic encryption, oblivious transfer (OT), etc. An instantiation based on Paillier encryption with communication complexity $\mathcal{O}(\kappa)$ is provided in [12] (for details see [12]).

Semi-honest Secure Triple Generation Protocol. The protocol Π_{TRIPLE} (see Figure 15) uses the two party protocol Π_{MULT} as a sub-protocol and allows a set of parties $\mathcal{X} \subset \mathcal{P}$ to generate one $\langle\cdot\rangle$-shared multiplication triplet $(\langle a\rangle_{\mathcal{X}}, \langle b\rangle_{\mathcal{X}}, \langle c\rangle_{\mathcal{X}})$. The protocol is executed assuming semi-honest adversary. The protocol is based on the following idea: every party $P_i \in \mathcal{X}$ selects a random a_i and b_i and commits the same. Then we set a and b to be the sum of all a_is and b_is. For setting c as $a \cdot b$, every pair of parties $P_i, P_j \in \mathcal{X}$ need to securely compute the "cross-terms" $a_i \cdot b_j$ and $a_j \cdot b_i$, for which they execute two instances

of Π_{MULT}. Once P_i computes its c_i, it publicly commits the same. Instantiating the calls to Π_{MULT} with that of [12] (based on the Paillier encryption), protocol Π_{TRIPLE} has communication complexity of $\mathcal{O}(|\mathcal{X}|^2\kappa)$ and $\mathcal{BC}\big(|\mathcal{X}|\kappa, |\mathcal{X}|\big)$.

Protocol Π_{TRIPLE}

The public input to the protocol is a set of parties $\mathcal{X} \subset \mathcal{P}$ containing at least one honest party. For the session id sid, every party $P_i \in \mathcal{X}$ participates with (sid, i) and does the following:

- Randomly select shares a_i, b_i.
- For all $P_j \in \mathcal{X} \setminus P_i$, run $\Pi_{\text{MULT}}(a_i, b_j) \rightarrow (d_{ij}, e_{ji})$ as party 1.
- For all $P_j \in \mathcal{X} \setminus P_i$, run $\Pi_{\text{MULT}}(a_j, b_i) \rightarrow (d_{ji}, e_{ij})$ as party 2.
- Set $c_i = a_i \cdot b_i + \sum_{P_j \in \mathcal{X} \setminus P_i} d_{ij} + \sum_{P_j \in \mathcal{X} \setminus P_i} e_{ij}$.
- Randomly select q_i, r_i, s_i, t_i, u_i and v_i and compute the commitments $\mathbf{C}_{a_i,q_i,r_i} = \mathsf{Comm}_{\mathsf{ck}}(a_i; q_i, r_i), \mathbf{C}_{b_i,s_i,t_i} = \mathsf{Comm}_{\mathsf{ck}}(b_i; s_i, t_i)$ and $\mathbf{C}_{c_i,u_i,v_i} = \mathsf{Comm}_{\mathsf{ck}}(c_i; u_i, v_i)$. Call $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, i, \mathbf{C}_{a_i,q_i,r_i}, \mathbf{C}_{b_i,s_i,t_i}, \mathbf{C}_{c_i,u_i,v_i}, \mathcal{X})$.
- Corresponding to each $P_j \in \mathcal{X}$, receive $(\mathsf{sid}, i, j, \mathbf{C}_{a_j,q_j,r_j}, \mathbf{C}_{b_j,s_j,t_j}, \mathbf{C}_{c_j,u_j,v_j})$ from $\mathcal{F}_{\text{BC}}$.
- Set $\langle a\rangle_i = (a_i, q_i, r_i, \{\mathbf{C}_{a_j,q_j,r_j}\}_{P_j \in \mathcal{X}}), \langle b\rangle_i = (b_i, s_i, t_i, \{\mathbf{C}_{b_j,s_j,t_j}\}_{P_j \in \mathcal{X}})$ and $\langle c\rangle_i = (c_i, u_i, v_i, \{\mathbf{C}_{c_j,u_j,v_j}\}_{P_j \in \mathcal{X}})$. Output $(\mathsf{sid}, i, \langle a\rangle_i, \langle b\rangle_i, \langle c\rangle_i)$ and halt.

Fig. 15. Protocol for Generating One $\langle\cdot\rangle$-shared Multiplication Triple Assuming No Active Corruptions.

Functionality for Generating $\langle\cdot\rangle$-sharing of a Random Value. Functionality $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$ (presented in Figure 16) generates an $\langle\cdot\rangle$-shared random value within a designated set of parties $\mathcal{X} \subset \mathcal{P}$, where each party in $\mathcal{X}$ "contributes" its "part" of the share and opening information for the shared random value.

Functionality $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$

The functionality interacts with a designated set of parties $\mathcal{X} \subset \mathcal{P}$ containing at least one honest party and the adversary $\mathcal{S}$ and is parametrized by the commitment key ck of a double-trapdoor commitment scheme. For the session id sid, the functionality does the following:

- On receiving $(\mathsf{sid}, i, s_i, u_i, v_i)$ from each party $P_i \in \mathcal{X}$, compute $s = \sum_{P_i \in \mathcal{X}} s_i, u = \sum_{P_i \in \mathcal{X}} u_i$ and $v = \sum_{P_i \in \mathcal{X}} v_i$ and $\mathbf{C}_{s,u,v} = \mathsf{Comm}_{\mathsf{ck}}(s; u, v)$. In addition, for each $P_i \in \mathcal{X}$, compute $\mathbf{C}_{s_i,u_i,v_i} = \mathsf{Comm}_{\mathsf{ck}}(s_i; u_i, v_i)$. Finally send $(\mathsf{sid}, i, \mathbf{C}_{s,u,v}, \{\mathbf{C}_{s_j,u_j,v_j}\}_{P_j \in \mathcal{X}})$ to every $P_i \in \mathcal{X}$ and halt.

Fig. 16. Functionality for Generating $\langle\cdot\rangle$-shared Random Value for a Designated Set $\mathcal{X} \subset \mathcal{P}$ with Dishonest-majority.

In [12], a realization of $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$ based on UC-secure multi-party commitment scheme was presented in the common reference string (CRS) model. The UC secure multi-party commitment scheme is further constructed using a CCA-secure encryption and the double-trapdoor homomorphic commitment scheme introduced in Section 2. Specifically, the following was shown; we refer to [12] for the details of the instantiation of $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$.

Lemma 6 ([12]). *Assuming CCA-secure encryption and double-trapdoor homomorphic commitment scheme, it is possible to (κ, s)-securely realize $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$ in the $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}})$-hybrid model in the UC framework. The protocol generates ℓ $\langle\cdot\rangle$-shared random values and has communication complexity $\mathcal{BC}\big(|\mathcal{X}|(\ell + s)\kappa, |\mathcal{X}|\big)$.*

Protocol for Reconstructing $\langle\cdot\rangle$-shared Value. Protocol $\Pi_{\text{REC}\langle\cdot\rangle}$ takes as input an $\langle\cdot\rangle$-sharing, say $\langle s\rangle_{\mathcal{X}}$ and either allows the honest parties in $\mathcal{X}$ to robustly reconstruct s or ensures that the honest parties in $\mathcal{X}$ can (locally) identify at least one corrupted party in $\mathcal{X}$. The protocol is based on the following standard idea:

let $\langle s\rangle_i = (s_i, u_i, v_i, \{\mathbf{C}_{s_j,u_j,v_j}\}_{P_j\in\mathcal{X}})$ be the information available to party $P_i \in \mathcal{X}$ corresponding to $\langle s\rangle_{\mathcal{X}}$. Then each P_i broadcasts s_i, u_i, v_i to the parties in $\mathcal{X}$ via $\mathcal{F}_{\text{BC}}$. Let each party P_i receive $\bar{s}_j, \bar{u}_j, \bar{v}_j$ from P_j. P_i then verifies if $\mathbf{C}_{s_j,u_j,v_j} = \mathsf{Comm}_{\mathsf{ck}}(\bar{s}_j; \bar{u}_j, \bar{v}_j)$. If the verification fails, P_i *identifies* P_j to be corrupted and outputs $(\mathsf{Failure}, i, j)$; otherwise P_i sums up all the shares to obtain s and outputs $(\mathsf{Success}, i, s)$. In the rest of the description, we will say that *the parties in $\mathcal{X}$ participate in $\Pi_{\text{REC}\langle\cdot\rangle}$ with $\langle s\rangle_{\mathcal{X}}$ and each $P_i \in \mathcal{X}$ outputs either* $(\mathsf{Success}, i, s)$ *or* $(\mathsf{Failure}, i, j)$ to mean the above. The protocol has communication complexity $\mathcal{BC}\big(|\mathcal{X}|\kappa, |\mathcal{X}|\big)$.

Beaver's Multiplication Protocol. Protocol $\Pi_{\text{BEA}}(\langle x\rangle_{\mathcal{X}}, \langle y\rangle_{\mathcal{X}}, \langle a\rangle_{\mathcal{X}}, \langle b\rangle_{\mathcal{X}}, \langle c\rangle_{\mathcal{X}})$ is a standard protocol for securely computing $\langle x \cdot y\rangle_{\mathcal{X}}$ from $\langle x\rangle_{\mathcal{X}}$ and $\langle y\rangle_{\mathcal{X}}$, at the cost of two public reconstruction. The protocol assumes that the parties in $\mathcal{X} \subset \mathcal{P}$ have access to an $\langle\cdot\rangle_{\mathcal{X}}$-shared random multiplication triple (a, b, c) unknown to the adversary, with $c = a \cdot b$. The protocol is based on the principle that $x \cdot y = (x - a + a) \cdot (y - b + b) = de + db + ae + c$, where $d = (x - a)$ and $e = (y - b)$. Hence if the parties in $\mathcal{X}$ reconstruct d and e, then they can locally compute $\langle x \cdot y\rangle_{\mathcal{X}} = de + d \cdot \langle b\rangle_{\mathcal{X}} + e \cdot \langle a\rangle_{\mathcal{X}} + \langle c\rangle_{\mathcal{X}}$. The security of x and y follows even after the reconstruction of d and e, as x and y are masked by random and private a and b respectively. To reconstruct d and e, the parties in $\mathcal{X}$ first locally compute $\langle d\rangle_{\mathcal{X}} = \langle x - a\rangle_{\mathcal{X}}$ and $\langle e\rangle_{\mathcal{X}} = \langle y - b\rangle_{\mathcal{X}}$, followed by invoking $\Pi_{\text{REC}\langle\cdot\rangle}$ with inputs $\langle d\rangle_{\mathcal{X}}$ and $\langle e\rangle_{\mathcal{X}}$. Depending on whether the instances of $\Pi_{\text{REC}\langle\cdot\rangle}$ are successful or not, an honest party in $\mathcal{X}$ may output $(\mathsf{Success}, i, \langle x \cdot y\rangle_i)$ or $(\mathsf{Failure}, i, j)$. In the rest of the description, we will say that *the parties in $\mathcal{X}$ participate in $\Pi_{\text{BEA}}(\langle x\rangle_{\mathcal{X}}, \langle y\rangle_{\mathcal{X}}, \langle a\rangle_{\mathcal{X}}, \langle b\rangle_{\mathcal{X}}, \langle c\rangle_{\mathcal{X}})$ and each P_i output either* $(\mathsf{Success}, i, \langle x \cdot y\rangle_i)$ *or* $(\mathsf{Failure}, i, j)$ to mean the above. The protocol has communication complexity $\mathcal{BC}\big(|\mathcal{X}|\kappa, |\mathcal{X}|\big)$.

D.1 The Preparation Stage of Protocol $\Pi_{\mathsf{C}}^{\text{NR}}$

We are now ready to discuss the preparation stage of our protocol $\Pi_{\mathsf{C}}^{\text{NR}}$. We pursue the same outline as followed by the preparation stage of the MPC protocol of [12] and describe the same briefly below. This is followed by the required adaptations in our context. The preparation stage of [12] provides security with abort. Namely the protocol generates the required $\langle\cdot\rangle$-shared triplets if all the parties behave honestly; otherwise if the honest parties identify any wrong-doing then they simply abort.

- **Triple Generation:** The involved parties generate many random $\langle\cdot\rangle$-shared triplets by executing many instances of Π_{TRIPLE}, assuming no active corruptions.
- **Verification of the Triples via Cut-and-choose:** A random fraction of the triplets are verified via cut-and-choose to detect any cheating attempts. Specifically, a random subset of generated triplets are selected and the parties are asked to disclose the randomness that they used in the instances of Π_{TRIPLE} for generating the selected triplets. If any cheating is detected then the involved parties abort, otherwise they proceed to the next step. If the test passes then with high probability it is ensured that the majority of the remaining untested triplets are "good" in the sense that they are honestly generated.
- **Proof of Knowledge:** The goal of this test is to ensure that for each remaining triplet, every party has the knowledge of their shares, thus ensuring independence required for UC security. More specifically, during the generation of an untested triplet, a corrupted party P_i could broadcast an arbitrary $\mathbf{C}_{a_i,\star,\star}, \mathbf{C}_{b_i,\star,\star}$ or $\mathbf{C}_{c_i,\star,\star}$, being oblivious to a_i, b_i and c_i. This is prevented by the following steps: First parties generate random $\langle\cdot\rangle$-shared values (by calling $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$) and then they open the difference of the triplets and those random shared values via protocol $\Pi_{\text{REC}\langle\cdot\rangle}$. Opening these differences is indeed a very simple proof of knowledge (see [12]). A cheating is detected if some of the opening fail. In that case the involved parties abort, otherwise they proceed to the next step.

– **Verification of the Triplets via Sacrificing Trick:** At this stage, the remaining triplets are verified for correctness via the well-known "sacrificing" trick [13]. Namely for every pair of remaining shared triplets (a, b, c) and (x, y, z), the parties generate a random r and recompute an $\langle\cdot\rangle$-sharing of $a \cdot b$, by assuming rx, ry, r^2z as a multiplication triplet; protocol Π_{BEA} is used for the same. Ideally if (a, b, c) and (x, y, z) are multiplication triplets, then the difference of the sharing of c and the recomputed ab should be a sharing of zero, which is verified by the parties publicly (using protocol $\Pi_{\text{REC}\langle\cdot\rangle}$). If any cheating is detected then the parties abort, else they proceed to the next step after discarding (x, y, z), whose security is sacrificed during the verification of (a, b, c). It follows that if the test passes then except with probability $1/p$ over the choice of r, the triplet (a, b, c) is indeed a correct multiplication triplet (see [12] for the details).
– **Privacy Amplification:** At this stage, the parties jointly perform privacy amplification and "distill" $\mathsf{C_M} + \mathsf{C_R}$ fully random private triplets from a set of $\mathcal{O}((\mathsf{C_M} + \mathsf{C_R}) + X)$ triplets, where X of them might not be private[11]; recall that $\mathsf{C_M}$ and $\mathsf{C_R}$ are the number of multiplication and random gates respectively in the circuit C. For this, $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$ along with Π_{BEA} is used. If any cheating is detected during Π_{BEA}, then the parties abort.

In our context, it is not enough to abort when a wrong-doing is detected. If some party $P_i \in \mathcal{X}$ identifies any party $P_j \in \mathcal{X}$ cheating in any of the steps for preparation stage, P_i alarms the parties in $\mathcal{P}$ by raising a complaint against P_j. This allows the parties in $\mathcal{P}$ to localize the fault to a pair of parties (P_i, P_j). To simplify the fault-localization, we set a designated party $P_{\mathsf{Ref}} \in \mathcal{X}$ with the smallest index Ref as the *referee* to locally identify any fault and report the same to the parties in $\mathcal{P}$. The fault localization step in each stage of the preparation stage is emphasized below.

– **Fault Localization During the Verification of the Triples via Cut-and-choose:** The parties in $\mathcal{X}$ first run the steps for the cut-and-choose triple-verification as in [12]. If any party P_i locally identifies any fault then it raises an alarm for the parties in $\mathcal{P}$. On receiving the alarm, every party in $\mathcal{X}$ broadcasts (to the parties in $\mathcal{X}$) their entire view (including the randomness used) in the generation of the triplets under testing. The referee P_{Ref} then "recomputes" every message a party $P_i \in \mathcal{X}$ should send to every other party $P_j \in \mathcal{X}$ and compares them with what P_i claims to send and what P_j claimed to receive. In case there is any mis-match, then P_{Ref} raises a complaint against both P_i and P_j among $\mathcal{P}$ and urges P_i and P_j to respond. Now depending upon the response, the parties can localize the fault to either (P_{Ref}, P_i) or (P_{Ref}, P_j) or (P_i, P_j). The important observation is that fault will never be localized to a pair of honest parties from $\mathcal{X}$. This is because the property of Π_{MULT} ensures that if both the participating parties are honest then they never conflict with each other. A located pair will contain at least one corrupted party.
– **Fault Localization in Proof of Knowledge:** The parties in $\mathcal{X}$ execute the same steps as in [12] for proving the knowledge of their shares. If any party $P_i \in \mathcal{X}$ locally identifies any fault during the instances of $\Pi_{\text{REC}\langle\cdot\rangle}$ (used to open the differences of triplets and random shared values), then P_i raises an alarm among the set $\mathcal{P}$, while the referee P_{Ref} is assigned the task of publicly reporting the identity of the party P_j it has caught cheating. The fault is then localized to (P_j, P_{Ref}). If an honest P_i raises an alarm, but a corrupted P_{Ref} does not identify any cheater, then the fault is localized to (P_i, P_{Ref}). It is easy to note that a located pair will contain at least one corrupted party.
– **Fault Localization During the Verification of the Triplets via Sacrificing Trick:** Here the parties in $\mathcal{X}$ first apply the sacrificing trick on each pair of remaining triplets. Now there are *three* situations under which a party $P_i \in \mathcal{X}$ can detect a fault. **(a)** The instances of Π_{BEA} is unsuccessful. In this case, the

[11] For the specific instantiation of Π_{MULT} based on Paillier encryption, this is indeed the case if one of the participating parties in Π_{MULT} is corrupted; see [12] for the details.

parties in $\mathcal{P}$ localize the fault in the same way as in the previous step. Namely P_i raises an alarm while P_{Ref} is asked to identify the cheating party. **(b)** The instances of $\Pi_{\text{REC}\langle\cdot\rangle}$ to open the difference of ab and c fails; the fault-localization in this case is also the same as in the previous step. **(c)** The difference of ab and c is non-zero. Clearly in this case, at least one of the involved triplet (in the pair) is not generated correctly and so the parties in $\mathcal{P}$ perform the fault-localization in the same way as in the cut-and-choose step. Namely, all the parties in $\mathcal{X}$ publicly open (to the parties in $\mathcal{X}$) their entire view produced during the generation of the two triplets and P_{Ref} is then asked to find a pair of "conflicting" parties.

– **Fault Localization in Privacy Amplification:** The parties in $\mathcal{X}$ execute the steps for privacy amplification [12]. If any cheating is detected by a party $P_i \in \mathcal{X}$ during the involved instances of Π_{BEA}, then the parties in $\mathcal{P}$ perform the fault localization in the same way as it is done during for a failed instance of Π_{BEA} in the previous stage.

The protocol steps for the preparation stage are given in Figure 17, where we give the formal steps for the fault localization with respect to only the first two phases; the formal steps for the fault localization for the remaining phases is not provided to avoid repetition.

In the protocol, B and λ are two parameters. In [12], it was shown that their preparation stage provides a statistical security of $2^{-B\log_2(1+\lambda)}$. They set B and λ as $B = 3.6s$ and $\lambda = 1/4$ to achieve a statistical security of 2^{-s}. Since our preparation stage is almost the same as that of [12] bar the fault-localization steps (which does not affect the statistical security at all), it follows easily via [12] that our preparation stage also provides a statistical security of $2^{-B\log_2(1+\lambda)}$. Intuitively this is due to the following reason: define a triplet to be a *good* one if the adversary could open it correctly during the **Cut-and-choose** step and make an honest party accept (this implies that such a triplet is generated honestly), otherwise call the triplet a *bad* triplet (i.e. such triplets are not generated honestly and so adversary may know some information about honest partys' shares for such triplets). Then it follows from [12] that if the protocol reaches the **Privacy Amplification** phase, then the probability that the triplets considered during this phase has more than B bad (and hence non-private) triplets is at most $(1+\lambda)^{-B}$. As a result, adversary may know at most B points on the polynomials $F(\cdot)$ and $G(\cdot)$ of degree at most d, implying $\mathsf{C_M} + \mathsf{C_R}$ degree of freedom in the view of the adversary. Note that as suggested in [12], instead of creating "big" polynomials $F(\cdot)$ and $G(\cdot)$ of huge degrees, we can partition the remaining triplets in $\mathcal{M}$ into batches of smaller size and accordingly use many polynomials of small degree, without affecting the security properties; we prefer to present the **Privacy Amplification** phase the way presented in [12].

Preparation Stage of $\Pi_{\mathsf{C}}^{\mathbf{NR}}$

The public input to the protocol is a set of parties $\mathcal{X} \subset \mathcal{P}$ containing at least one honest party and a referee $P_{\mathsf{Ref}} \in \mathcal{X}$, with the smallest index Ref. For the session id sid, every party $P_i \in \mathcal{P}$ participates with (sid, i) and does the following:

Triple-generation Assuming No Active Corruption — If $P_i \in \mathcal{X}$ then participate in the protocol Π_{TRIPLE} $(1+\lambda)(4(\mathsf{C_M}+\mathsf{C_R})+4B-2)$ times to generate a set $\mathcal{M}$ of $(1+\lambda)(4(\mathsf{C_M}+\mathsf{C_R})+4B-2)$ $\langle\cdot\rangle$-shared triplets.

Testing the Triplets via Cut-and-Choose — If $P_i \in \mathcal{X}$ then do the following:

- Call $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$ to sample a random string str that determines a subset $\mathcal{T} \subset \mathcal{M}$ of size $\lambda(4(\mathsf{C_M}+\mathsf{C_R})+4B-2)$. Set $\mathcal{M} = \mathcal{M} \setminus \mathcal{T}$. Let $\mathsf{View}_i^{\mathcal{T}}$ denote the randomness used by P_i and the messages received from the other parties in $\mathcal{X}$, during the instances of Π_{TRIPLE} used for generating the triplets in $\mathcal{T}$. Reveal $\mathsf{View}_i^{\mathcal{T}}$ to the parties in $\mathcal{X}$ by calling $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, i, \mathsf{View}_i^{\mathcal{T}}, \mathcal{X})$.
- Corresponding to each $P_j \in \mathcal{X}$, receive $(\mathsf{sid}, i, j, \mathsf{View}_j^{\mathcal{T}})$ from $\mathcal{F}_{\text{BC}}$. Using $\{\mathsf{View}_j^{\mathcal{T}}\}_{P_j \in \mathcal{X}}$, reproduce every message that should have been sent by every sender $P_a \in \mathcal{X}$ to every receiver $P_b \in \mathcal{X}$ during the generation of the triplets in $\mathcal{T}$, and compare it with the corresponding value that the recipient P_b claims to have received. If any conflict is detected, then do the following for the smallest indexed conflicting parties P_a, P_b:
  - If $P_i \neq P_{\mathsf{Ref}}$, then call $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, i, \texttt{Err}, \mathcal{P})$ to indicate to the parties in $\mathcal{P}$ that a conflict has been detected.
  - Else call $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, \mathsf{Ref}, \texttt{Err}, P_a, P_b, l, x, \overline{x}, \mathcal{P})$ to indicate that referee P_i identified $P_a, P_b \in \mathcal{X}$ the least indexed conflicting parties and a message with index l where P_a should have sent x but P_b claimed to receive $\overline{x} \neq x$.
- If the message $(\mathsf{sid}, i, \mathsf{Ref}, \texttt{Err}, P_a, P_b, l, x, \overline{x})$ is received from $\mathcal{F}_{\text{BC}}$ and if $P_a = P_i$ or $P_b = P_i$, then call $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, i, \mathsf{Agree}, P_{\mathsf{Ref}}, \mathcal{P})$ to indicate that you agree with P_{Ref}, else call $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, i, \mathsf{Disagree}, P_{\mathsf{Ref}}, \mathcal{P})$.

Fault Localization —

- If the message $(\mathsf{sid}, i, \mathsf{Ref}, \texttt{Err}, P_a, P_b, l, x, \overline{x})$ is received from $\mathcal{F}_{\text{BC}}$ and subsequently **(a)** if $(\mathsf{sid}, i, a, \mathsf{Disagree}, P_{\mathsf{Ref}})$ is received from $\mathcal{F}_{\text{BC}}$, then output $(\mathsf{sid}, i, \mathsf{Failure}, P_{\mathsf{Ref}}, P_a)$ and halt **(b)** if $(\mathsf{sid}, i, b, \mathsf{Disagree}, P_{\mathsf{Ref}})$ is received from $\mathcal{F}_{\text{BC}}$, then output $(\mathsf{sid}, i, \mathsf{Failure}, P_{\mathsf{Ref}}, P_b)$ and halt. Else output $(\mathsf{sid}, i, \mathsf{Failure}, P_a, P_b)$ and halt.
- If no message of the form $(\mathsf{sid}, i, \mathsf{Ref}, \texttt{Err}, \star, \star, \star, \star, \star)$ is received from $\mathcal{F}_{\text{BC}}$, but corresponding to some $P_j \in \mathcal{X}$ the message $(\mathsf{sid}, i, j, \texttt{Err})$ is received from $\mathcal{F}_{\text{BC}}$, then output $(\mathsf{sid}, i, \mathsf{Failure}, P_{\mathsf{Ref}}, P_j)$ and halt.

Proof of Knowledge — If $P_i \in \mathcal{X}$ then do the following for every (untested) triplet $(\langle a\rangle_{\mathcal{X}}, \langle b\rangle_{\mathcal{X}}, \langle c\rangle_{\mathcal{X}})$ in $\mathcal{M}$: Sample three random $\langle\cdot\rangle$-shared values $\langle r\rangle_{\mathcal{X}}, \langle s\rangle_{\mathcal{X}}, \langle u\rangle_{\mathcal{X}}$ by invoking $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$. Participate in instances of $\Pi_{\text{REC}\langle\cdot\rangle}$ with $\langle r-a\rangle_{\mathcal{X}}, \langle s-b\rangle_{\mathcal{X}}$ and $\langle u-c\rangle_{\mathcal{X}}$. If $(\mathsf{sid}, \mathsf{Failure}, i, j)$ is the output in any of the instances of $\Pi_{\text{REC}\langle\cdot\rangle}$, then do the following:

- If $P_i \neq P_{\mathsf{Ref}}$ then call $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, i, j, \texttt{Err}, \mathcal{P})$ to indicate that a cheating has been detected.
- Else if $P_i = P_{\mathsf{Ref}}$ then call $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, \mathsf{Ref}, \texttt{Err}, j, \mathcal{P})$ to indicate P_j is identified as a cheater; if there are several such P_js then select the one with the minimum index j.

Fault Localization —If a message $(\mathsf{sid}, i, \mathsf{Ref}, \texttt{Err}, j, \mathcal{P})$ is received from $\mathcal{F}_{\text{BC}}$, then output $(\mathsf{sid}, i, \mathsf{Failure}, P_{\mathsf{Ref}}, P_j)$ and halt. Else if a message $(\mathsf{sid}, i, j, \texttt{Err})$ is received from $\mathcal{F}_{\text{BC}}$, then output $(\mathsf{sid}, i, \mathsf{Failure}, P_i, P_j)$ and halt.

Verification Via sacrificing Trick — If $P_i \in \mathcal{X}$ then do the following for every pair of triplets $(\langle a\rangle_{\mathcal{X}}, \langle b\rangle_{\mathcal{X}}, \langle c\rangle_{\mathcal{X}})$ and $(\langle x\rangle_{\mathcal{X}}, \langle y\rangle_{\mathcal{X}}, \langle z\rangle_{\mathcal{X}})$ in $\mathcal{M}$: Call $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$ and sample a random[a] r. Participate in $\Pi_{\text{BEA}}(\langle a\rangle_{\mathcal{X}}, \langle b\rangle_{\mathcal{X}}, \langle rx\rangle_{\mathcal{X}}, \langle ry\rangle_{\mathcal{X}}, \langle r^2 z\rangle_{\mathcal{X}})$ for computing $\langle \overline{c}\rangle_{\mathcal{X}}$ followed by participation in $\Pi_{\text{REC}\langle\cdot\rangle}$ with $\langle \overline{c}-c\rangle_{\mathcal{X}}$. If no cheating has been identified during Π_{BEA}, $\Pi_{\text{REC}\langle\cdot\rangle}$ and if $\overline{c}-c=0$, then store $(\langle a\rangle_{\mathcal{X}}, \langle b\rangle_{\mathcal{X}}, \langle c\rangle_{\mathcal{X}})$ for future use and drop $(\langle x\rangle_{\mathcal{X}}, \langle y\rangle_{\mathcal{X}}, \langle z\rangle_{\mathcal{X}})$ from $\mathcal{M}$. Else proceed to the fault-localization step.

Fault Localization — If the parties in $\mathcal{X}$ have raised a complaint due to the failure of Π_{BEA} or $\Pi_{\text{REC}\langle\cdot\rangle}$, then localize the fault in the same way as in the case of fault-localization for the **Proof of Knowledge** step. Else localize the fault in the same way as in the **Cut-and-Choose** step by asking the parties in $\mathcal{X}$ to open their entire view of the disputed triplet.

Privacy Amplification — The parties in $\mathcal{X}$ are now left with $2(\mathsf{C_M}+\mathsf{C_R})+2B-1$ triplets $\{(\langle a^k\rangle_{\mathcal{X}}, \langle b^k\rangle_{\mathcal{X}}, \langle c^k\rangle_{\mathcal{X}})\}_{k=1,\ldots,2(\mathsf{C_M}+\mathsf{C_R})+2B-1}$ in $\mathcal{M}$. Let $d=(\mathsf{C_M}+\mathsf{C_R})+B-1$. If $P_i \in \mathcal{X}$ then do the following:

- Invoke $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$ $2(d+1)$ times to generate $\langle f^{(1)}\rangle_{\mathcal{X}}, \ldots, \langle f^{(d+1)}\rangle_{\mathcal{X}}$ and $\langle g^{(1)}\rangle_{\mathcal{X}}, \ldots, \langle g^{(d+1)}\rangle_{\mathcal{X}}$.
- Let $F(\cdot)$ and $G(\cdot)$ be the polynomials of degree at most d such that $F(\alpha_k)=f^{(k)}$ and $G(\alpha_k)=g^{(k)}$ for $k=1,\ldots,d+1$. Locally compute $\langle F(\alpha_{d+2})\rangle_{\mathcal{X}}, \ldots, \langle F(\alpha_{2d+1})\rangle_{\mathcal{X}}$ and $\langle G(\alpha_{d+2})\rangle_{\mathcal{X}}, \ldots, \langle G(\alpha_{2d+1})\rangle_{\mathcal{X}}$. For $k=1,\ldots,2d+1$, participate in $\Pi_{\text{BEA}}(\langle F(\alpha_k)\rangle_{\mathcal{X}}, \langle G(\alpha_k)\rangle_{\mathcal{X}}, \langle a^{(k)}\rangle_{\mathcal{X}}, \langle b^{(k)}\rangle_{\mathcal{X}}, \langle c^{(k)}\rangle_{\mathcal{X}})$ for computing $\langle h^{(k)}\rangle_{\mathcal{X}} = \langle F(\alpha_k)\cdot G(\alpha_k)\rangle_{\mathcal{X}}$.
- If any cheating is identified during Π_{BEA}, then proceed to the fault localization step. Else let $H(\cdot)$ be the polynomial of degree at most $2d$ such that $H(\alpha_i)=h^{(i)}$ for $i=1,\ldots,2d+1$. Then output $(\mathsf{sid}, i, \mathsf{Success}, \{(\langle \mathbf{a}^{(k)}\rangle_{\mathcal{X}}, \langle \mathbf{b}^{(k)}\rangle_{\mathcal{X}}, \langle \mathbf{c}^{(k)}\rangle_{\mathcal{X}})\}_{k=1,\ldots,\mathsf{C_M}+\mathsf{C_R}})$ and halt, where $\mathbf{a}^{(k)}=F(-\alpha_k), \mathbf{b}^{(k)}=G(-\alpha_k)$ and $\mathbf{c}^{(k)}=H(-\alpha_k)$.

Fault Localization — If any complaint is raised due to the failure of Π_{BEA}, then localize the fault as in the **Proof of Knowledge** step. Else every $P_i \in \mathcal{P} \setminus \mathcal{X}$ output $(\mathsf{sid}, i, \mathsf{Success})$ and halt.

[a] It is enough to sample a *single* r for all the pairs of available triplets.

Fig. 17. Generating $\mathsf{C_M}+\mathsf{C_R}$ $\langle\cdot\rangle$-shared Multiplication Triples with Statistical Security $2^{-B\log_2(1+\lambda)}$.

D.2 Protocol $\Pi_{\mathsf{C}}^{\mathbf{NR}}$

In this section the protocol $\Pi_{\mathsf{C}}^{\mathrm{NR}}$ is presented in Figure 18, where during the circuit-evaluation stage, we follow the idea outlined earlier in section 3.2. Note that during the circuit evaluation, an instance of Π_{BEA} may fail, in which case the parties in $\mathcal{P}$ localize the fault via the referee P_{Ref} in the same way as it was done in the preparation stage.

Protocol $\Pi_{\mathsf{C}}^{\mathbf{NR}}$

The public input to the protocol is a set of parties $\mathcal{X} \subset \mathcal{P}$ containing at least one honest party and an arithmetic circuit C over $\mathbb{F}_p$ consisting of in input gates, out output gates, $\mathsf{C_M}$ multiplication gates and $\mathsf{C_R}$ random gates. In addition, $\langle x_1 \rangle_{\mathcal{X}}, \ldots, \langle x_{\mathsf{in}} \rangle_{\mathcal{X}}$ are the $\langle \cdot \rangle$-shared inputs for C. Let $P_{\mathsf{Ref}} \in \mathcal{X}$ be the party with the smallest index Ref who is set as the referee to localize any fault occurred during the protocol.

For the session id sid, party $P_i \in \mathcal{P}$ participates with (sid, i) and does the following:

Preparation Stage: execute the steps of Figure 17.

Computation Stage: If $(\mathsf{sid}, i, \mathsf{Success}, \{(\langle \mathbf{a}^{(k)} \rangle_{\mathcal{X}}, \langle \mathbf{b}^{(k)} \rangle_{\mathcal{X}}, \langle \mathbf{c}^{(k)} \rangle_{\mathcal{X}})\}_{k=1,\ldots,\mathsf{C_M}+\mathsf{C_R}})$ or $(\mathsf{sid}, i, \mathsf{Success})$ is obtained at the end of preparation stage, then do the following:

- **$\langle \cdot \rangle$-shared Evaluation of the Circuit** C — If $P_i \in \mathcal{X}$ then do the following for every gate in the circuit C:
  - **Input Gate**: For $l = 1, \ldots, \mathsf{in}$, associate $\langle x_l \rangle_{\mathcal{X}}$ with the corresponding input gate of C.
  - **Random Gate**: For the k^{th} random gate in C where $k \in \{1, \ldots, \mathsf{C_R}\}$, associate $\langle \mathbf{a}^{(k)} \rangle_{\mathcal{X}}$ as the output of the random gate.
  - **Addition Gate**: If $\langle x \rangle_{\mathcal{X}}$ and $\langle y \rangle_{\mathcal{X}}$ are the $\langle \cdot \rangle$-shared inputs of the gate, then locally compute $\langle x + y \rangle_{\mathcal{X}} = \langle x \rangle_{\mathcal{X}} + \langle y \rangle_{\mathcal{X}}$ and associate it as the output of the addition gate.
  - **Multiplication Gate**: For the k^{th} multiplication gate in C with the $\langle \cdot \rangle$-shared inputs $\langle x \rangle_{\mathcal{X}}$ and $\langle y \rangle_{\mathcal{X}}$ where $k \in \{1, \ldots, \mathsf{C_M}\}$, associate the triplet $(\langle \mathbf{a}^{(\mathsf{C_R}+k)} \rangle_{\mathcal{X}}, \langle \mathbf{b}^{(\mathsf{C_R}+k)} \rangle_{\mathcal{X}}, \langle \mathbf{c}^{(\mathsf{C_R}+k)} \rangle_{\mathcal{X}})$. Participate in $\Pi_{\mathrm{BEA}}(\langle x \rangle_{\mathcal{X}}, \langle y \rangle_{\mathcal{X}}, \langle \mathbf{a}^{(\mathsf{C_R}+k)} \rangle_{\mathcal{X}}, \langle \mathbf{b}^{(\mathsf{C_R}+k)} \rangle_{\mathcal{X}}, \langle \mathbf{c}^{(\mathsf{C_R}+k)} \rangle_{\mathcal{X}})$ to compute $\langle x \cdot y \rangle_{\mathcal{X}}$. If $(\mathsf{sid}, i, \mathsf{Failure}, j)$ with $P_j \in \mathcal{X}$ is obtained during the instance of Π_{BEA} then do the following:
    - If $P_i \neq P_{\mathsf{Ref}}$ then call $\mathcal{F}_{\mathrm{BC}}$ with $(\mathsf{sid}, i, \mathtt{Err}, \mathcal{P})$ to indicate that a cheating has been detected while executing Π_{BEA}.
    - Else if $P_i = P_{\mathsf{Ref}}$ then call $\mathcal{F}_{\mathrm{BC}}$ with $(\mathsf{sid}, \mathsf{Ref}, \mathtt{Err}, j, \mathcal{P})$ to indicate that P_j is identified as a cheater while executing Π_{BEA}; if there are several such P_js then select the one with the minimum index j.
- **Fault Localization** —
  - If there exists a multiplication gate in C corresponding to which a message $(\mathsf{sid}, i, \mathsf{Ref}, \mathtt{Err}, j)$ is received on the behalf of P_{Ref} from $\mathcal{F}_{\mathrm{BC}}$ then output $(\mathsf{sid}, i, \mathsf{Failure}, P_{\mathsf{Ref}}, P_j)$ and halt.
  - Else if there exists a multiplication gate in C corresponding to which a message $(\mathsf{sid}, i, j, \mathtt{Err})$ is received from $\mathcal{F}_{\mathrm{BC}}$ on the behalf of $P_j \in \mathcal{X}$, but no message of the form $(\mathsf{sid}, i, \mathsf{Ref}, \mathtt{Err}, \star, \star)$ is received from $\mathcal{F}_{\mathrm{BC}}$ on the behalf of P_{Ref}, then output $(\mathsf{sid}, i, \mathsf{Failure}, P_{\mathsf{Ref}}, P_j)$ and halt.
  - Else if $P_i \in \mathcal{P} \setminus \mathcal{X}$ then output $(\mathsf{sid}, i, \mathsf{Success})$ and halt; otherwise output $(\mathsf{sid}, i, \mathsf{Success}, \langle y_1 \rangle_{\mathcal{X}}, \ldots, \langle y_{\mathsf{out}} \rangle_{\mathcal{X}})$ and halt, where $\langle y_1 \rangle_{\mathcal{X}}, \ldots, \langle y_{\mathsf{out}} \rangle_{\mathcal{X}}$ are the $\langle \cdot \rangle$-shared outputs associated with the output gates of C.

Fig. 18. Protocol for Secure $\langle \cdot \rangle$-shared Evaluation of a Given Circuit C with Statistical Security $1 - 2^{-B \log_2(1+\lambda)}$.

The correctness of the protocol follows via the binding property of the commitment and the detailed informal discussion above, while we appeal to [12] for the proof of privacy in UC secure framework. We now prove Lemma 4 (the lemma statement is available in Section 3), by setting $\lambda = 1/4$ and $B = 3.6s$ as done in [12], so that the protocol provides a statistical security of 2^{-s}.

Proof of Lemma 4: We prove the communication complexity of the preparation stage, with the observation that $\mathsf{C_M} + \mathsf{C_R} = \mathcal{O}(|\mathsf{C}|)$. During the **Triple-generation** phase, $\mathcal{O}(\mathsf{C_M} + \mathsf{C_R} + B)$ instances of Π_{TRIPLE} are executed by the parties in $\mathcal{X}$, thus requiring communication complexity of $\mathcal{O}(|\mathcal{X}|^2(|\mathsf{C}| + B)\kappa)$ and $\mathcal{BC}\big(|\mathcal{X}|(|\mathsf{C}| + B)\kappa, |\mathcal{X}|\big)$.

During the **Cut-and-Choose** phase, $\mathcal{O}(\mathsf{C_M} + \mathsf{C_R} + B)$ calls to $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$ are made for generating $\mathcal{O}(\mathsf{C_M} + \mathsf{C_R} + B)$ random $\langle\cdot\rangle$-shared commitments with statistical security $2^{-B\log_2(1+\lambda)}$, incurring communication complexity of $\mathcal{BC}\big(|\mathcal{X}|(|\mathsf{C}| + B)\kappa, |\mathcal{X}|\big)$. In addition, the parties in $\mathcal{X}$ need to broadcast among themselves their entire view of Π_{TRIPLE} with respect to $\mathcal{O}(\mathsf{C_M} + \mathsf{C_R} + B)$ triplets. This incurs a communication complexity of $\mathcal{BC}\big(|\mathcal{X}|^2(|\mathsf{C}| + B)\kappa, |\mathcal{X}|\big)$. During the fault-localization step, the parties in $\mathcal{X}$ need to broadcast $\mathcal{O}(\kappa)$ bits to the parties in $\mathcal{P}$, thus requiring communication complexity of $\mathcal{BC}\big(|\mathcal{X}|\kappa, n\big)$.

During the **Proof of Knowledge** phase, $\mathcal{O}(\mathsf{C_M} + \mathsf{C_R} + B)$ calls to $\mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}$ are made and $\mathcal{O}(\mathsf{C_M} + \mathsf{C_R} + B)$ instances of $\Pi_{\text{REC}\langle\cdot\rangle}$ are executed by the parties in $\mathcal{X}$, thus requiring a communication complexity of $\mathcal{BC}\big(|\mathcal{X}|(|\mathsf{C}|+B)\kappa, |\mathcal{X}|\big)$. In addition, during the fault-localization step, the parties in $\mathcal{X}$ need to broadcast $\mathcal{O}(\kappa)$ bits to the parties in $\mathcal{P}$, thus requiring communication complexity of $\mathcal{BC}\big(|\mathcal{X}|\kappa, n\big)$.

During the **Correctness** phase, $\mathcal{O}(\mathsf{C_M} + \mathsf{C_R} + B)$ instances of Π_{BEA} and $\Pi_{\text{REC}\langle\cdot\rangle}$ are executed by the parties in $\mathcal{X}$. In addition, the parties in $\mathcal{X}$ may need to publicly open among themselves the entire view of Π_{TRIPLE} with respect to a disputed pair of triplet. Thus this phase has communication complexity of $\mathcal{BC}\big(|\mathcal{X}|(|\mathsf{C}| + B)\kappa, |\mathcal{X}|\big)$, with an additional communication complexity of $\mathcal{BC}\big(|\mathcal{X}|\kappa, n\big)$ for the fault-localization step. It follows easily that the **Privacy Amplification** phase as well as the circuit evaluation stage has communication complexity of $\mathcal{BC}\big(|\mathcal{X}|(|\mathsf{C}| + B)\kappa, |\mathcal{X}|\big)$ for executing the steps within $\mathcal{X}$ and has communication complexity of $\mathcal{BC}\big(|\mathcal{X}|\kappa, n\big)$ for any possible fault-localization.

During the computation stage, $\mathsf{C_M}$ instances of Π_{BEA} are executed and fault-localization is done at most once. It thus follows that setting $B = 3.6s$, the protocol has communication complexity $\mathcal{O}(|\mathcal{X}|^2(|\mathsf{C}| + s)\kappa), \mathcal{BC}\big(|\mathcal{X}|^2(|\mathsf{C}| + s)\kappa, |\mathcal{X}|\big)$ and $\mathcal{BC}\big(|\mathcal{X}|\kappa, n\big)$. □

E Proof of Theorem 1

Security. We prove the security by designing a simulator for the protocol Π_f. Let $T \subset \mathcal{P}$ be the set of parties under the control of $\mathcal{A}$ during the protocol Π_f; we present a simulator $\mathcal{S}_f$ (interacting with the functionality $\mathcal{F}_f$) for $\mathcal{A}$ in Figure 19. The high level idea for the simulator is the following: the simulator takes the input $\{x^{(i)}\}_{P_i \in T}$ and interacts with $\mathcal{F}_f$ to obtain the function output y. The simulator then invokes $\mathcal{A}$ with the inputs $\{x^{(i)}\}_{P_i \in T}$ and simulates each message that $\mathcal{A}$ would have received in the protocol Π_f from the honest parties and from the functionalities called therein, step by step. Notice that the simulator $\mathcal{S}_f$ also needs to simulate the protocol steps of the honest parties for the sub-protocols $\Pi_{[\cdot]\to\langle\cdot\rangle}$, $\Pi_{\langle\cdot\rangle\to[\cdot]}$, $\Pi^{\text{NR}}_{\mathsf{ckt}_l}$ and $\Pi_{\text{RANDZERO}[\cdot]}$. Specifying the simulator steps for these subprotocols would make the description of $\mathcal{S}_f$ complicated. So for the ease of presentation, we define three sub-simulators $\mathcal{S}_{[\cdot]\to\langle\cdot\rangle}$ (Fig. 20), $\mathcal{S}_{\langle\cdot\rangle\to[\cdot]}$ (Fig. 21), and $\mathcal{S}_{\text{RANDZERO}[\cdot]}$ (Fig. 22) which are invoked by $\mathcal{S}_f$ for simulating the steps of the honest parties for the instances of $\Pi_{[\cdot]\to\langle\cdot\rangle}$, $\Pi_{\langle\cdot\rangle\to[\cdot]}$ and $\Pi_{\text{RANDZERO}[\cdot]}$ respectively; technically, the steps specified for $\mathcal{S}_{[\cdot]\to\langle\cdot\rangle}, \mathcal{S}_{\langle\cdot\rangle\to[\cdot]}$ and $\mathcal{S}_{\text{RANDZERO}[\cdot]}$ are actually done by the main simulator $\mathcal{S}_f$. While invoking these "sub-simulators", $\mathcal{S}_f$ will provide its entire internal state to them and the sub-simulators then return back their internal state (after the required simulation) to the main simulator. Similarly, we also assume the presence of a simulator $\mathcal{S}^{\text{NR}}_{\mathsf{ckt}_l}$, which can be invoked by $\mathcal{S}_f$ to simulate the steps of the honest parties for the protocol $\mathcal{S}^{\text{NR}}_{\mathsf{ckt}_l}$. We do not explicitly give the steps of $\mathcal{S}^{\text{NR}}_{\mathsf{ckt}_l}$, but rather appeal to the simulator of the MPC protocol of [12] because the protocol steps of $\Pi^{\text{NR}}_{\mathsf{ckt}_l}$ are almost the same as the MPC protocol of [12], bar the fault-localization steps. However, simulating the steps of fault-localization is straight forward, since the simulator will know the entire states of all the honest parties in $\Pi^{\text{NR}}_{\mathsf{ckt}_l}$ and so any wrong-doings by the corrupted parties can be easily identified by the simulator exactly as it was identified by an honest party in $\Pi^{\text{NR}}_{\mathsf{ckt}_l}$.

It is easy to show that $\text{IDEAL}_{\mathcal{F}_f,\mathcal{S}_f,\mathcal{Z}} \overset{c}{\approx} \text{REAL}_{\Pi_f,\mathcal{A},\mathcal{Z}}$ in the $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{COMMITTEE}}, \mathcal{F}_{\text{GEN}[\cdot]}, \mathcal{F}_{\text{GENRAND}\langle\cdot\rangle}, \mathcal{F}_{\text{ZK.BC}})$-hybrid settings due to the privacy of the the secret sharing schemes and the statistical hiding prop-

Simulator $\mathcal{S}_f$

The simulator plays the role of the honest parties and simulates each step of the protocol Π_f as follows. The communication of the $\mathcal{Z}$ with the adversary $\mathcal{A}$ is handled as follows: Every input value received by the simulator from $\mathcal{Z}$ is written on $\mathcal{A}$'s input tape. Likewise, every output value written by $\mathcal{A}$ on its output tape is copied to the simulator's output tape (to be read by the environment $\mathcal{Z}$). The simulator then does the following for the session ID sid:

Initialization. $\mathcal{S}_f$ sets its internal variables $\mathcal{L} = \mathcal{P}$, $\mathfrak{n} = n$, $\mathfrak{t} = t$ and $\mathsf{NewCom} = 1$.

CRS Generation. On receiving (sid, i) from every $P_i \in T$, simulator $\mathcal{S}_f$, on behalf of $\mathcal{F}_{\text{CRS}}$, computes $\mathsf{Gen}(1^\kappa) \rightarrow (\mathsf{ck}, \tau_0, \tau_1)$ and $G(1^\kappa) \rightarrow (\mathsf{pk}, \mathsf{sk})$, sets $\mathsf{CRS} = (\mathsf{ck}, \mathsf{pk})$ and sends (sid, i, CRS) to every $P_i \in T$.

Input commitment. On behalf of every honest party $P_i \in \mathcal{P} \setminus T$, $\mathcal{S}_f$ picks three random polynomials over $\mathbb{F}_p$, $f^{(i)}(\cdot), g^{(i)}(\cdot), h^{(i)}(\cdot)$ of degree t such that $f^{(i)}(0) = 0$ and imitates the behavior of the honest parties. That is, $\mathcal{S}_f$ computes the commitment $\mathbf{C}_{f^{(i)}(0),g^{(i)}(0),h^{(i)}(0)} = \mathsf{Comm}_{\mathsf{ck}}(f^{(i)}(0); g^{(i)}(0), h^{(i)}(0))$ and sends $(\mathsf{sid}, i, \mathbf{C}_{f^{(i)}(0),g^{(i)}(0),h^{(i)}(0)})$ to every corrupted $P_j \in T$ on behalf of $\mathcal{F}_{\text{BC}}$. When a corrupted $P_i \in T$ invokes $\mathcal{F}_{\text{BC}}$ with $(\mathsf{sid}, i, \mathbf{C}_{f^{(i)}(0),g^{(i)}(0),h^{(i)}(0)}, \mathcal{P})$, simulator $\mathcal{S}_f$ acts on behalf of $\mathcal{F}_{\text{BC}}$ and sends $\mathbf{C}_{f^{(i)}(0),g^{(i)}(0),h^{(i)}(0)}$ to every $P_j \in T$.

$[\cdot]$-sharing of Inputs. For every honest $P_i \in \mathcal{P} \setminus T$, simulator $\mathcal{S}_f$ acts on behalf of functionality $\mathcal{F}_{\text{GEN}[\cdot]}$ with $(\mathsf{sid}, i, f^{(i)}(\cdot), g^{(i)}(\cdot), h^{(i)}(\cdot))$ and hands $(\mathsf{sid}, j, i, [f^{(i)}(0)]_j)$ to every $P_j \in T$. Then for every corrupted $P_i \in T$, on receiving $(\mathsf{sid}, i, f^{(i)}(\cdot), g^{(i)}(\cdot), h^{(i)}(\cdot))$ from P_i (as the dealer), $\mathcal{S}_f$, on behalf of $\mathcal{F}_{\text{GEN}[\cdot]}$, sends $(\mathsf{sid}, j, i, [f^{(i)}(0)]_i)$ to every $P_j \in T$, after verifying the polynomials $f^{(i)}(\cdot), g^{(i)}(\cdot), h^{(i)}(\cdot)$ with respect to the corresponding commitment $\mathbf{C}_{f^{(i)}(0),g^{(i)}(0),h^{(i)}(0)}$ (as done by the functionality $\mathcal{F}_{\text{GEN}[\cdot]}$). Locally, simulator maintains the following information:

- $\mathcal{S}_f$ stores the input of corrupted $P_i \in T$ as $x^{(i)} = f^{(i)}(0)$, where $f^{(i)}(\cdot)$ is received from corrupted P_i. Further it sets the input of honest $P_i \in \mathcal{P} \setminus T$ as $x^{(i)} = 0$.
- For every $P_i \in \mathcal{P}$, it stores the entire $[x^{(i)}]$.

$\mathcal{S}_f$ hands $\{x^{(i)}\}_{P_i \in T}$ to the MPC functionality $\mathcal{F}_f$ on behalf of the corrupted parties and gets back the outputs y from the functionality. Next $\mathcal{S}_f$ computes the remaining circuit using 0s as the inputs of the honest parties and $\{x^{(i)}\}_{P_i \in T}$ as the inputs of the corrupted parties. For these inputs, it knows the value to be associated with each wire of the circuit. Thus it knows the circuit output $\bar{y}$ resulted from the above set of inputs, namely 0s as the inputs of the honest parties and $\{x^{(i)}\}_{P_i \in T}$ as the inputs of the corrupted parties.

Start of while loop over the sub-circuits. Set $l = 1$ and while $l < L$, $\mathcal{S}_f$ continues as follows:

- **Committee Selection.** If $\mathsf{NewCom} = 1$, on receiving $(\mathsf{sid}, i, \mathcal{L})$ from every party $P_i \in T$, $\mathcal{S}_f$ on behalf of $\mathcal{F}_{\text{COMMITTEE}}$ picks $\mathfrak{c}$ parties from its local set $\mathcal{L}$ at random and assigns them to $\mathcal{C}$. It then sends $(\mathsf{sid}, P_i, \mathcal{C})$ to every $P_i \in T$.
- **$[\cdot]$ to $\langle\cdot\rangle_{\mathcal{C}}$ Conversion of Inputs of ckt_l.** Let $[x_1], \ldots, [x_{\mathsf{in}_l}]$ denote $[\cdot]$-sharing of the inputs to the sub-circuit ckt_l. For $k \in \{1, \ldots, \mathsf{in}_l\}$, $\mathcal{S}_f$ invokes the sub-simulator $\mathcal{S}_{[\cdot]\rightarrow\langle\cdot\rangle}$ (Fig. 20) that simulates the steps of the honest parties in $\Pi_{[\cdot]\rightarrow\langle\cdot\rangle}$, with $(\mathsf{sid}, \{[x_k]_i\}_{P_i \in \mathcal{P}\setminus T}, \mathcal{C})$ (namely with the shares corresponding to the honest parties). The sub-simulator returns $\mathcal{S}_f$ with $(\mathsf{sid}, \{\langle x_k\rangle_i\}_{P_i \in \mathcal{C} \wedge (\mathcal{P}\setminus T)})$.
- **Evaluation of the Sub-circuit ckt_l.** The simulator $\mathcal{S}_f$ invokes the simulator $\mathcal{S}^{\text{NR}}_{\mathsf{ckt}_l}$ (namely the simulator of the MPC protocol of [12] with the appropriate modifications in our context to do fault localization) for simulating the steps of the honest parties in the protocol $\Pi^{\text{NR}}_{\mathsf{ckt}_l}$.
- **$\langle\cdot\rangle_{\mathcal{C}}$ to $[\cdot]$ conversion of Outputs of ckt_l.** $\mathcal{S}_f$ invokes $\mathcal{S}_{\langle\cdot\rangle\rightarrow[\cdot]}$ with $(\mathsf{sid}, \{\langle y_k\rangle_i\}_{P_i \in \mathcal{C}\wedge(\mathcal{P}\setminus T)}, \mathcal{C})$ for every $k \in \{1, \ldots, \mathsf{out}_l\}$ and gets back either $(\mathsf{sid}, \{[y_k]_i\}_{P_i \in \mathcal{P}\setminus T})$ or $(\mathsf{sid}, i, \mathsf{Failure}, P_a, P_b)$ or $(\mathsf{sid}, i, \mathsf{Failure}, P_a)$ and does the following:
  - If $(\mathsf{sid}, \{[y_k]_i\}_{P_i \in \mathcal{P}\setminus T})$ is received for every k, increment $l = l + 1$, set $\mathsf{NewCom} = 0$, store the sharings and return to the while loop.
  - If $(\mathsf{sid}, i, \mathsf{Failure}, P_a, P_b)$ is received for *some* $k \in \mathsf{out}_l$, update $\mathcal{L}$ as $\mathcal{L} = \mathcal{L} \setminus \{P_a, P_b\}$, $\mathfrak{t}$ as $\mathfrak{t} = \mathfrak{t} - 1$, $\mathfrak{n}$ as $\mathfrak{n} = \mathfrak{n} - 2$.
  - If $(\mathsf{sid}, i, \mathsf{Failure}, P_a)$ is received for *some* $k \in \mathsf{out}_l$, update $\mathcal{L}$ as $\mathcal{L} = \mathcal{L} \setminus \{P_a\}$, $\mathfrak{t}$ as $\mathfrak{t} = \mathfrak{t} - 1$, $\mathfrak{n}$ as $\mathfrak{n} = \mathfrak{n} - 1$.
  - Set $\mathsf{NewCom} = 1$ and go to **Committee Selection Step**.

Output Rerandomization Let $[\bar{y}]$ denote the $[\cdot]$-sharing of the output of ckt. $\mathcal{S}_f$ invokes $\mathcal{S}_{\text{RANDZERO}[\cdot]}$ with input $(\mathsf{sid}, y - \bar{y})$. $\mathcal{S}_{\text{RANDZERO}[\cdot]}$ simulates the honest parties in protocol $\Pi_{\text{RANDZERO}[\cdot]}$ and returns to $\mathcal{S}_f$ $(\mathsf{sid}, \{[y - \bar{y}]_i\}_{P_i \in (\mathcal{P}\setminus T)})$. $\mathcal{S}_f$ locally computes $[y]_i = [\bar{y}]_i + [y - \bar{y}]_i$ for every $P_i \in \mathcal{P} \setminus T$.

Output Computation. On behalf of every honest P_i, $\mathcal{S}_f$ sends $(\mathsf{sid}, i, j, f_i, g_i, h_i)$ to every $P_j \in T$ where $[y]_i = (f_i, g_i, h_i, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j \in \mathcal{P}})$. Clearly every $P_i \in T$ will recover y at the end due to the output rerandomization step.

The simulator then outputs $\mathcal{A}$'s output and terminate.

Fig. 19. Simulator for the adversary $\mathcal{A}$ corrupting at most t parties in the set $T \subset \mathcal{P}$ in the protocol Π_f.

erty of the underlying commitment scheme. For the correctness of the protocol, we rely on the trapdoor security and binding properties of the underlying double trapdoor commitment scheme.

Simulator $\mathcal{S}_{[\cdot]\to\langle\cdot\rangle}$

For session id sid, on receiving $(\mathsf{sid}, \{[s]_i\}_{P_i\in\mathcal{P}\setminus T}, \mathcal{X})$ from $\mathcal{S}_f$, $\mathcal{S}_{[\cdot]\to\langle\cdot\rangle}$ interacts with the corrupted parties in T on behalf of the honest parties in an instance of Protocol $\Pi_{[\cdot]\to\langle\cdot\rangle}$. The simulator $\mathcal{S}_{[\cdot]\to\langle\cdot\rangle}$ is aware of the internal state of the honest parties corresponding to the $[s]$. The simulator proceeds as follows:

Verifiably $\langle\cdot\rangle$-sharing the Share and Opening Information in $[s]_i$. First, $\mathcal{S}_{[\cdot]\to\langle\cdot\rangle}$ acts on behalf of every honest P_i as the dealer in an instance of $\Pi_{\langle\cdot\rangle}$ with $(\mathsf{sid}, [s]_i, \mathcal{X})$ such that $[s]_i = (f_i, g_i, h_i, \{\mathbf{C}_{f_j,g_j,h_j}\}_{P_j\in\mathcal{P}})$. It also simulates every honest party P_i in an instance of $\Pi_{\langle\cdot\rangle}$ where a corrupted $P_k \in T$ acts as a dealer.

Identifying the Correctly $\langle\cdot\rangle$-shared Shares of s and Generating $\langle s\rangle_{\mathcal{X}}$. If a corrupted $P_i \in T$ is caught cheating during conflict resolution step, then exclude it from a set $\mathcal{H}$ that is initialized to $\mathcal{P}$. Otherwise, for every corrupted $P_i \in T$, let it receive $\langle f_i\rangle_k$ on behalf of every honest $P_k \in (\mathcal{P}\setminus T) \wedge \mathcal{X}$. Without loss of generality, let[a] $\mathcal{H} = \{P_1, \ldots, P_{|\mathcal{H}|}\}$ and let $c_1, \ldots, c_{|\mathcal{H}|}$ be the publicly known Lagrange interpolation coefficients, such that $c_1 f_1 + \ldots + c_{|\mathcal{H}|} f_{|\mathcal{H}|} = s$. Then the simulator locally computes $\langle s\rangle_i = c_1\langle f_1\rangle_i + \ldots + c_{|\mathcal{H}|}\langle f_{|\mathcal{H}|}\rangle_i$ on behalf of every *honest* $P_i \in \mathcal{X}$ and returns $(\mathsf{sid}, \{\langle s\rangle_i\}_{P_i\in\mathcal{X}\wedge(\mathcal{P}\setminus T)})$ to $\mathcal{S}_f$.

[a] The set $\mathcal{H}$ will be of size more than $t+1$.

Fig. 20. Simulator $\mathcal{S}_{[\cdot]\to\langle\cdot\rangle}$ to be Invoked by the MPC Simulator $\mathcal{S}_f$ for Simulating the Steps of Sub-protocol $\Pi_{[\cdot]\to\langle\cdot\rangle}$ in Π_f

Simulator $\mathcal{S}_{\langle\cdot\rangle\to[\cdot]}$

For session id sid, on receiving $(\mathsf{sid}, \{\langle s\rangle_i\}_{P_i\in\mathcal{X}\wedge(\mathcal{P}\setminus T)}, \mathcal{X})$ from $\mathcal{S}_f$, $\mathcal{S}_{\langle\cdot\rangle\to[\cdot]}$ interacts with the corrupted parties in T on behalf of the honest parties in an instance of Protocol $\Pi_{\langle\cdot\rangle\to[\cdot]}$. Note that the simulator is aware of the internal state of the honest parties corresponding to the $\langle s\rangle$. The simulator proceeds as follows:

- $\mathcal{S}_{\langle\cdot\rangle\to[\cdot]}$, on behalf of every *honest* $P_i \in \mathcal{X}$, interprets $\langle s\rangle_i$ as $(s_i, u_i, v_i, \{\mathbf{C}^{P_i}_{s_j,u_j,v_j}\}_{P_j\in\mathcal{X}})$ and sends $(\mathsf{sid}, k, i, \{\mathbf{C}^{P_i}_{s_j,u_j,v_j}\}_{P_j\in\mathcal{X}})$ to every $P_k \in T$ (acting on behalf of functionality $\mathcal{F}_{\mathrm{BC}}$ that would have been called by P_i in the hybrid protocol). On receiving $(\mathsf{sid}, i, \{\overline{\mathbf{C}^{P_i}_{s_j,u_j,v_j}}\}_{P_j\in\mathcal{X}}, \mathcal{P})$ from every $P_i \in T$, $\mathcal{S}_{\langle\cdot\rangle\to[\cdot]}$ acts on behalf of $\mathcal{F}_{\mathrm{BC}}$ and sends $(\mathsf{sid}, k, i, \{\overline{\mathbf{C}^{P_i}_{s_j,u_j,v_j}}\}_{P_j\in\mathcal{X}})$ to every $P_k \in T$.
- If there exists a pair of parties $P_a, P_b \in \mathcal{X}$, such that P_a is honest and $P_b \in T$ and $\{\mathbf{C}^{P_a}_{s_j,u_j,v_j}\}_{P_j\in\mathcal{X}} \neq \{\overline{\mathbf{C}^{P_b}_{s_j,u_j,v_j}}\}_{P_j\in\mathcal{X}}$, then return $(\mathsf{sid}, \mathsf{Failure}, P_a, P_b)$ to $\mathcal{S}_f$ and halt.
- On behalf of every *honest* $P_i \in \mathcal{X}$, $\mathcal{S}_{\langle\cdot\rangle\to[\cdot]}$ acts as a D and selects $f^{(i)}(\cdot), g^{(i)}(\cdot)$ and $h^{(i)}(\cdot)$ such that they are random polynomials of degree at most t, subject to the condition that $f^{(i)}(0) = s_i, g^{(i)}(0) = u_i$ and $h^{(i)}(0) = v_i$. Then on behalf of $\mathcal{F}_{\mathrm{GEN}[\cdot]}$, $\mathcal{S}_{\langle\cdot\rangle\to[\cdot]}$ creates the $[s_i]$ exactly in the way $\mathcal{F}_{\mathrm{GEN}[\cdot]}$ would compute on input $(\mathsf{sid}, i, f^{(i)}(\cdot), g^{(i)}(\cdot), h^{(i)}(\cdot))$ from P_i as the dealer. Then it hands $(\mathsf{sid}, k, i, [s_i]_k)$ to every $P_k \in T$. On receiving $(\mathsf{sid}, i, \overline{f^{(i)}}(\cdot), \overline{g^{(i)}}(\cdot), \overline{h^{(i)}}(\cdot))$ from a corrupted $P_i \in T$ acting as D, $\mathcal{S}_{\langle\cdot\rangle\to[\cdot]}$ acts exactly as $\mathcal{F}_{\mathrm{GEN}[\cdot]}$ and returns either $(\mathsf{sid}, i, k, \mathsf{Failure})$ or $(\mathsf{sid}, i, k, [s_k]_i)$ to every $P_i \in T$. $\mathcal{S}_{\langle\cdot\rangle\to[\cdot]}$ returns $(\mathsf{sid}, \mathsf{Failure}, P_k)$ to $\mathcal{S}_f$ when $(\mathsf{sid}, i, k, \mathsf{Failure})$ was generated for any $P_k \in T$. Otherwise, it locally computes $[s]_i = \sum_{P_k\in\mathcal{X}}[s_k]_i$ for every honest $P_i \in \mathcal{P}\setminus T$, returns $(\mathsf{sid}, \{[s]_i\}_{P_i\in Parties\setminus T})$ to $\mathcal{S}_f$ and halts.

Fig. 21. Simulator $\mathcal{S}_{\langle\cdot\rangle\to[\cdot]}$ to be Invoked by the MPC Simulator $\mathcal{S}_f$ for Simulating the Steps of Sub-protocol $\Pi_{\langle\cdot\rangle\to[\cdot]}$ in Π_f

Simulator $\mathcal{S}_{\text{RANDZERO}[\cdot]}$

For the session id sid, on receiving $(\mathsf{sid}, y - \bar{y})$ from $\mathcal{S}_f$, $\mathcal{S}_{\text{RANDZERO}[\cdot]}$ interacts with the corrupted parties in T on behalf of the honest parties in an instance of Protocol $\Pi_{\text{RANDZERO}[\cdot]}$. The simulator proceeds as follows:

Publicly Committing 0:

- On behalf of honest party P_h (it just chooses any honest party from the set $\mathcal{P}$) randomly selects $u_h, v_h \in \mathbb{F}_p$, sets $r_h = y - \bar{y}$ and computes $\mathbf{C}_{r_h,u_i,v_i} = \mathsf{Comm}_{\mathsf{ck}}(y - \bar{y}; u_i, v_i)$. On behalf of every other honest party P_i, it randomly selects $u_i, v_i \in \mathbb{F}_p$, sets $r_i = 0$ and computes $\mathbf{C}_{r_i,u_i,v_i} = \mathsf{Comm}_{\mathsf{ck}}(r_i; u_i, v_i)$. On behalf of $\mathcal{F}_{\text{ZK.BC}}$ corresponding to every honest P_i, it then sends $(\mathsf{sid}, i, \mathbf{C}_{r_i,u_i,v_i})$ to every $P_j \in T$. On receiving $(\mathsf{sid}, i, \mathbf{C}_{r_i,u_i,v_i}, u_i, v_i)$ from every corrupted $P_i \in T$, it acts as $\mathcal{F}_{\text{ZK.BC}}$ and verifies if $\mathbf{C}_{r_i,u_i,v_i} = \mathsf{Comm}_{\mathsf{ck}}(0; u_i, v_i)$. It the tests passes, then the simulator on behalf of $\mathcal{F}_{\text{ZK.BC}}$ sends $(\mathsf{sid}, i, \mathbf{C}_{r_i,u_i,v_i})$ to every $P_j \in T$. Otherwise, it sends $(\mathsf{sid}, i, \perp)$ to every $P_j \in T$.
- It then constructs a set $\mathcal{T}$, initialized to $\emptyset$ and include in $\mathcal{T}$ all the honest parties in $\mathcal{P}$ and $P_i \in T$ if $\mathbf{C}_{r_i,u_i,v_i} = \mathsf{Comm}_{\mathsf{ck}}(0; u_i, v_i)$ was true for P_i.

$[\cdot]$**-sharing** 0:

- On behalf of honest party P_i, it selects three random polynomials $f^{(i)}(\cdot), g^{(i)}(\cdot)$ and $h^{(i)}(\cdot)$ each of degree at most t, subject to the condition that $f^{(i)}(0) = r_i, g^{(i)}(0) = u_i$ and $h^{(i)}(0) = v_i$. On behalf of $\mathcal{F}_{\text{GEN}[\cdot]}$ for an honest P_i, it sends $(\mathsf{sid}, j, i, f_j^{(i)}, g_j^{(i)}, h_j^{(i)})$ to every $P_j \in T$. Then for every corrupted $P_i \in T$, on receiving $(\mathsf{sid}, i, f^{(i)}(\cdot), g^{(i)}(\cdot), h^{(i)}(\cdot))$ from P_i (as the dealer), $\mathcal{S}_{\text{RANDZERO}[\cdot]}$, on behalf of $\mathcal{F}_{\text{GEN}[\cdot]}$, sends $(\mathsf{sid}, j, i, [f^{(i)}(0)]_i)$ to every $P_j \in T$, after verifying the polynomials $f^{(i)}(\cdot), g^{(i)}(\cdot), h^{(i)}(\cdot)$ with respect to the corresponding commitment $\mathbf{C}_{r_i,u_i,v_i}$ (as done by the functionality $\mathcal{F}_{\text{GEN}[\cdot]}$). If the polynomials fails the test, remove P_i from $\mathcal{T}$.
- It locally computes $[y - \bar{y}]_i = \sum_{P_j \in \mathcal{T}} [r_j]_i$ and returns $(\mathsf{sid}, \{[y - \bar{y}]_i\}_{P_i \in \mathcal{P} \setminus T})$ to $\mathcal{S}_f$ and halt.

Fig. 22. Simulator for $\Pi_{\text{RANDZERO}[\cdot]}$

Between a Rock and a Hard Place: Interpolating Between MPC and FHE

A. Choudhury, J. Loftus, E. Orsini, A. Patra and N.P. Smart

Dept. Computer Science,
University of Bristol,
United Kingdom.
{Ashish.Choudhary,Emmanuela.Orsini,Arpita.Patra}@bristol.ac.uk,
{loftus,nigel}@cs.bris.ac.uk

Abstract. We present a computationally secure MPC protocol for threshold adversaries which is parametrized by a value L. When $L = 2$ we obtain a classical form of MPC protocol in which interaction is required for multiplications, as L increases interaction is reduced, in that one requires interaction only after computing a higher degree function. When L approaches infinity one obtains the FHE based protocol of Gentry, which requires no interaction. Thus one can trade communication for computation in a simple way. Our protocol is based on an interactive protocol for "bootstrapping" a somewhat homomorphic encryption (SHE) scheme. The key contribution is that our presented protocol is highly communication efficient enabling us to obtain reduced communication when compared to traditional MPC protocols for relatively small values of L.

1 Introduction

In the last few years computing on encrypted data via either Fully Homomorphic Encryption (FHE) or Multi-Party Computation (MPC) has been subject to a remarkable number of improvements. Firstly, FHE was shown to be possible [29]; and this was quickly followed by a variety of applications and performance improvements [9, 12, 11, 30, 31, 39, 40]. Secondly, whilst MPC has been around for over thirty years, only in the last few years we have seen an increased emphasis on practical instantiations; with some very impressive results [8, 22, 37].

We focus on MPC where n parties wish to compute a function on their respective inputs. Whilst the computational overhead of MPC protocols, compared to computing "in the clear", is relatively small (for example in practical protocols such as [25, 37] a small constant multiple of the "in the clear" cost), the main restriction on practical deployment of MPC is the communication cost. Even for protocols in the preprocessing model, evaluating arithmetic circuits over $\mathbb{F}_p$, the communication cost in terms of number of bits per multiplication gate and per party is a constant multiple of the bit length, $\log p$, of the data being manipulated for a typically large value of the constant. This is a major drawback of MPC protocols since communication is generally more expensive than computation. Theoretical results like [19] (for the computational case) and [20] (for the information theoretic case) bring down the per gate per party communication cost to a very small quantity; essentially $\mathcal{O}(\frac{\log n}{n} \cdot \log |C| \cdot \log p)$ bits for a circuit C of size $|C|$. While these results suggest that the communication cost can be asymptotically brought down to a constant for large n, the constants are known to be large for any practical purpose. Our interest lies in constructing efficient MPC protocols where the efficiency is measured in terms of *exact* complexity rather than the *asymptotic* complexity.

In his thesis, Gentry [28] showed how FHE can be used to reduce the communication cost of MPC down to virtually zero for any number of parties. In Gentry's MPC protocol all parties encrypt to each other their inputs under a shared FHE public key. They then compute the function homomorphically, and at the end perform a shared decryption. This implies an MPC protocol whose communication is limited to a function of the input and output sizes, and not to the complexity of the circuit. However, this reduction in communication complexity comes at a cost, namely the huge expense of evaluating homomorphically the function. With current understanding of FHE technology, this solution is completely infeasible in practice.

A variant of Gentry's protocol was presented by Asharov et al. in [1] where the parties outsource their computation to a server and only interact via a distributed decryption. The key innovation in [1] was that independently generated

This article is based on an earlier article: Asiacrypt 2013, IACR 2013, http://dx.doi.org/10.1007/978-3-642-42045-0_12.

(FHE) keys can be combined into a "global" FHE key with distributed decryption capability. We do not assume such a functionality of the keys (but one can easily extend our results to accommodate this); instead we focus on using distributed decryption to enable *efficient* multi-party bootstrapping. In addition the work of [1], in requiring an FHE scheme, as opposed to the SHE scheme of our work, requires the assumption of circular security of the underlying FHE scheme (and hence more assumptions).

In [25], following on the work in [7], the authors propose an MPC protocol which uses an SHE scheme as an "optimization". Based in the preprocessing model, the authors utilize an SHE scheme which can evaluate circuits of multiplicative depth one to optimize the preprocessing step of an essentially standard MPC protocol. The optimizations, and use of SHE, in [25] are focused on the case of computational improvements. In this work we invert the use of SHE in [25], by using it for the online phase of the MPC protocol, so as to optimize the communication efficiency for any number of parties.

In essence we interpolate between the two extremes of traditional MPC protocols (with high communication but low computational costs) and Gentry's FHE based solution (with high computation but low communication costs). Our interpolation is dependent on a parameter, which we label as L, where $L \geq 2$. At one extreme, for $L = 2$ our protocol resembles traditional MPC protocols, whilst at the other extreme, for $L = \infty$ our protocol is exactly that of Gentry's FHE based solution. We emphasize that our construction is general in that *any* SHE can be used which supports homomorphic computation of depth *two* circuits and threshold decryption. Thus the requirements on the underlying SHE scheme are much weaker than the previous SHE (FHE) based MPC protocols, such as the one by Asharov et al. [1], which relies on the specifics of LWE (learning with errors) based SHE i.e. *key-homomorphism* and demands homomorphic computation of depth L circuits for big enough L to bootstrap.

The solution we present is in the preprocessing model; in which we allow a preprocessing phase which can compute data which is neither input, nor function, dependent. This preprocessed data is then consumed in the online phase. As usual in such a model our goal is for efficiency in the online phase only. We present our basic protocol and efficiency analysis for the case of passive threshold adversaries only; i.e. we can tolerate up to t passive corruptions where $t < n$. We then note that security against t active adversaries with $t < n/3$ can be achieved for no extra cost in the online phase. For the active security case, essentially the same communication costs can be achieved even when $t < n/2$, bar some extra work (which is *independent* of $|C|$) to eliminate the cheating parties when they are detected. The security of our protocols are proven in the standard UC framework [13].

We note that our focus is on the MPC protocols providing *robustness* and *fairness*[1], which is impossible to achieve in general without assuming $t < n/2$ [15, 32]. Indeed in several real-life applications it may be desirable to have these properties. However we stress that we could deal with the dishonest majority setting (i.e. $t < n$) by utilizing additional zero-knowledge proof techniques to show that the distributed decryptions are performed correctly; however as our goal is to achieve low *exact* communication complexity (as opposed to low asymptotic complexity) we feel that such a discussion would deviate from the thrust of our work. In adding the corresponding associated proofs of correctness we would still achieve an asymptotic improvement in communication complexity over other MPC protocols with dishonest majority; but this is not our focus and so in the rest of the paper, we avoid discussing about the setting of dishonest majority.

Finally we note that our results on communication complexity, both in a practical and in an asymptotic sense, in the computational setting are comparable (if not better) than the best known results in the information theoretic and computational settings. Namely the best known optimally resilient statistically secure MPC protocol with $t < n/2$ has (asymptotic) communication complexity of $\mathcal{O}(n)$ per multiplication [5], whereas ours is $\mathcal{O}(n/L)$ (see Section 9 for the analysis of our protocol). With near optimal resiliency of $t < (\frac{1}{3} - \epsilon)n$, the best known perfectly secure MPC protocol has (asymptotic) communication complexity of $\mathcal{O}(\text{polylog } n)$ per multiplication [20], but a huge constant is hiding under the $\mathcal{O}$. In the computational settings, with near optimal resiliency of $t < (\frac{1}{2} - \epsilon)n$, the best known MPC protocol has (asymptotic) communication complexity of $\mathcal{O}(\text{polylog } n)$ per multiplication [19], but again a huge constant is hiding under the $\mathcal{O}$. All these protocols can not win over ours when *exact* communication complexity is compared for even small values of L.

[1] Informally robustness means that the adversary cannot deny the honest parties from obtaining the correct output, while fairness guarantees that either everyone receives the output or no one obtains the output.

Overview: Our protocol is intuitively simple. We first take an L-levelled SHE scheme (strictly it has $L+1$ levels, but can evaluate circuits with L levels of multiplications) which possesses a distributed decryption protocol for the specific access structure required by our MPC protocol. We assume that the SHE scheme is implemented over a ring which supports N embeddings of the underlying finite field $\mathbb{F}_p$ into the message space of the SHE scheme. Almost all known SHE schemes support such packing of the finite field into the plaintext slots in an SIMD manner [30, 40]; and such packing has been crucial in the implementation of SHE in various applications [21, 25, 31].

Clearly with such a setup we can implement Gentry's MPC solution for circuits of multiplicative depth L. All that remains is how to "bootstrap" from circuits with multiplicative depth L to arbitrary circuits. The standard solution would be to bootstrap the FHE scheme directly, following the blueprint outlined in Gentry's thesis. However, in the case of applications to MPC we could instead utilize a protocol to perform the bootstrapping. In a nutshell that is exactly what we propose.

The main issue then is show how to efficiently perform the bootstrapping in a distributed manner; where efficiency is measured in terms of computational and communication performance. Naively performing an MPC protocol to execute the bootstrapping phase will lead to a large communication overhead, due to the inherent overhead in dealing with homomorphic encryptions. But on its own this is enough to obtain our asymptotic interpolation between FHE and MPC; we however aim to provide an efficient and practical interpolation. That is one which is efficient for small values of L. It turns out that a special case of a suitable bootstrapping protocol can be found as a sub-procedure of the MPC protocol in [25]. We extract the required protocol, generalise it, and then apply it to our MPC situation.

To ease exposition we will not utilize the packing from [30] to perform evaluations of the depth L sub-circuits; we see this as a computational optimization which is orthogonal to the issues we will explore in this paper. In any practical instantiation of the protocol of this paper such a packing could be used, as described in [30], in evaluating the circuit of multiplicative depth L. However, we will use this packing to perform the bootstrapping in a communication efficient manner.

The bootstrapping protocol runs in two phases. In the first (offline) phase we repeatedly generate sets of ciphertexts, one set for each party, such that all parties learn the ciphertexts but only the given party learns their underlying messages (which are assumed to be packed). The offline phase can be run in either a passive, covert or active security model, irrespective of the underlying access structure of the MPC protocol following ideas from [22]. In the second (online) phase the data to be bootstrapped is packed together, a random mask is added (computed from the offline phase data), a distributed decryption protocol is executed to obtain the masked data which is then re-encrypted, the mask is subtracted and then the data is unpacked. All these steps are relatively efficient, with communication only being required for the distributed decryption.

To apply our interactive bootstrapping method efficiently we need to make a mild assumption on the circuit being evaluated; this is similar to the assumptions used in [19, 20, 26]. The assumption can be intuitively seen as saying that the circuit is relatively wide enough to enable packing of enough values which need to be bootstrapped at each respective level. We expect that most circuits in practice will satisfy our assumption, and we will call the circuits which satisfy our requirement "well formed".

We pause to note that the ability to open data within the MPC protocol enables one to perform more than a simple evaluation of an arithmetic circuit. This observation is well known in the MPC community, where it has been used to obtain efficient protocols for higher level functions [14, 18]. Thus enabling a distributed bootstrapping also enables one to produce more efficient protocols than purely FHE based ones.

We instantiate our protocol with the BGV scheme [10] and obtain sufficient parameter sizes following the methodology in [22, 31]. Due to the way we utilize the BGV scheme we need to restrict to MPC protocols for arithmetic circuits over a finite field $\mathbb{F}_p$, with $p \equiv 1 \pmod{m}$ with $m = 2 \cdot N$ and $N = 2^r$ for some r. The distributed decryption method uses a "smudging" technique which means that the modulus used in the BGV scheme needs to be larger than what one would need to perform just the homomorphic operations. Removing this smudging technique, and hence obtaining an efficient protocol for distributed decryption, for *any* SHE scheme is an interesting open problem; with many potential applications including that described in this paper.

We show that even for a very small value of L, in particular $L = 5$, we can achieve better communication efficiency than many practical MPC protocols in the preprocessing model. Most practical MPC protocols such as [8, 25, 37] require the transmission of at least two finite field elements per multiplication gate between each pair of parties. In [25] a technique is presented which can reduce this to the transmission of an average of three field elements per

multiplication gate per party (and not per pair of parties). Note the models in [8] (three party, one passive adversary) and [25, 37] (n party, dishonest majority, active security) are different from ours (we assume honest majority, active security); but even mapping these protocols to our setting of n party honest majority would result in the same communication characteristics. We show that for relatively small values of L, i.e. $L > 8$, one can obtain a communication efficiency of less than one field element per gate and party (details available in Section 9).

Clearly, by setting L appropriately one can obtain a communication efficiency which improves upon that in [19, 20]; albeit we are only interested in communication in the online phase of a protocol in the preprocessing model whilst [19, 20] discuss total communication cost over all phases. But we stress this is not in itself interesting, as Gentry's FHE based protocol can beat the communication efficiency of [19, 20] in any case. What is interesting is that we can beat the communication efficiency of the online phase of practical MPC protocols, with very small values of L indeed. Thus the protocol in this paper may provide a practical tradeoff between existing MPC protocols (which consume high bandwidth) and FHE based protocols (which require huge computation).

Our protocol therefore enables the following use-case: it is known that SHE schemes only become prohibitively computationally expensive for large L; indeed one of the reasons why the protocols in [22, 25] are so efficient is that they restrict to evaluating homomorphically circuits of multiplicative depth one. With our protocol parties can a priori decide the value of L, for a value which enables them to produce a computationally efficient SHE scheme. Then they can execute an MPC protocol with communication costs reduced by effectively a factor of L. Over time as SHE technology improves the value of L can be increased and we can obtain Gentry's original protocol. Thus our methodology enables us to interpolate between the case of standard MPC and the eventual goal of MPC with almost zero communication costs.

2 Well Formed Circuits

In this section we define what we mean by well formed circuits, and the pre-processing which we require on our circuits. We take as given an arithmetic circuit C defined over a finite field $\mathbb{F}_p$. In particular the circuit C is a directed acyclic graph consisting of edges made up of n_I input wires, n_O output wires, and n_W internal wires, plus a set of nodes being given by a set of gates $\mathbb{G}$. The gates are divided into sets of Add gates $\mathbb{G}_A$ and Mult gates $\mathbb{G}_M$, with $\mathbb{G} = \mathbb{G}_A \cup \mathbb{G}_M$, with each Add/Mult gate taking two wires (or a constant value in $\mathbb{F}_p$) as input and producing one wire as output. The circuit is such that all input wires are open on their input ends, and all output wires are open on their output ends, with the internal wires being connected on both ends. We let the depth of the circuit d be the length of the maximum path from an input wire to an output wire. Our definition of a well formed circuit is parametrized by two positive integer values N and L.

We now associate inductively to each wire in the circuit an integer valued label as follows. The input wires are given the label one; then all other wires are given a label according to the following rule (where we assume a constant input to a gate has label L)

$$\begin{aligned} \text{Label of output wire of Add gate } &= \min(\text{Label of input wires}), \\ \text{Label of output wire of Mult gate } &= \min(\text{Label of input wires}) - 1. \end{aligned}$$

Thus the minimum value of a label is $1 - d$ (which is negative for a general d). Looking ahead, the reason for starting with an input label of one is when we match this up with our MPC protocol this will result in low communication complexity for the input stage of the computation.

We now augment the circuit, to produce a new circuit C^{aug} which will have labels in the range $[1, \ldots, L]$, by adding in some special gates which we will call Refresh gates; the set of such gates are denoted as $\mathbb{G}_R$. A Refresh gate takes as input a maximum of N wires, and produces as output an exact copy of the specified input wires. The input requirement is that the input wires must have label in the range $[1, \ldots, L]$, and all that the Refresh gate does is relabel the labels of the gate's input wires to be L. At the end of the augmentation process we require the invariant that all wire labels in C^{aug} are then in the range $[1, \ldots, L]$, and the circuit is now essentially a collection of "sub-circuits" of multiplicative depth at most $L - 1$ glued together using Refresh gates. However, we require that this is done with as small a number of Refresh gates as possible.

Definition 1 (Well Formed Circuit). *A circuit C will be called well formed if the number of* Refresh *gates in the associated augmented circuit C^{aug} is at most $\frac{2 \cdot |\mathbb{G}_M|}{L \cdot N}$.*

We expect that "most" circuits will be well formed due to the following argument: We first note that the only gates which concern us are multiplication gates; so without loss of generality we consider a circuit C consisting only of multiplication gates. The circuit has d layers, and let the width of C (i.e. the number of gates) at layer i be w_i. Consider the algorithm to produce C^{aug} which considers each layer in turn, from $i = 1$ to d and adds Refresh gates where needed. When reaching level i in our algorithm to produce C^{aug} we can therefore assume (by induction) that all input wires at this layer have labels in the range $[1, \ldots, L]$. To maintain the invariant we only need to apply a Refresh operation to those input wires which have label one. Let p_i denote the proportion of wires at layer i which have label one when we perform this process. It is clear that the number of required Refresh gates which we will add into C^{aug} at level i will be at most $\lceil 2 \cdot p_i \cdot w_i / N \rceil$, where the factor of two comes from the fact that each multiplication gate has two input wires.

Assuming a large enough circuit we can assume for most layers that this proportion p_i will be approximately $1/L$, since wires will be refreshed after their values have passed through L multiplication gates. So summing up over all levels, the expected number of Refresh gates in C^{aug} will be:

$$\sum_{i=1}^{d} \left\lceil \frac{2 \cdot w_i}{L \cdot N} \right\rceil \approx \frac{2}{L \cdot N} \cdot \sum_{i=1}^{d} w_i = \frac{2 \cdot |\mathbb{G}_M|}{L \cdot N}.$$

Note, we would expect that for most circuits this upper bound on the number of Refresh gates could be easily met. For example our above rough analysis did not take into account the presence of gates with fan-out greater than one (meaning there are less wires to Refresh than we estimated above), nor did it take into account utilizing unused slots in the Refresh gates to refresh wires with labels not equal to one.

Determining an optimum algorithm for moving from C to a suitable C^{aug}, with a minimal number of Refresh gates, is an interesting optimization problem which we leave as an open problem; however clearly the above outlined greedy algorithm will work for most circuits.

3 Threshold L-Levelled Packed Somewhat Homomorphic Encryption (SHE)

In this section, we present a detailed explanation of the syntax and requirements for our Threshold L-Levelled Packed Somewhat Homomorphic Encryption Scheme. The scheme will be parametrized by a number of values; namely the security parameter κ, the number of levels L, the amount of packing of plaintext elements which can be made into one ciphertext N, a statistical security parameter sec (for the security of the distributed decryption) and a pair (t, n) which defines the threshold properties of our scheme. In practice the parameter N will be a function of L and κ. The message space of the SHE scheme is defined to be $\mathcal{M} = \mathbb{F}_p^N$, and we embed the finite field $\mathbb{F}_p$ into $\mathcal{M}$ via a map $\chi : \mathbb{F}_p \longrightarrow \mathcal{M}$. See Section 7 for a discussion as to the various choices one has for χ when we specialise to the BGV SHE scheme.

Let $\mathcal{C}(L)$ denote the family of circuits consisting of addition and multiplication gates whose labels follow the conventions in Section 2; except that input wires have label L and whose minimum wire label is zero. Thus $\mathcal{C}(L)$ is the family of standard arithmetic circuits of multiplicative depth at most L which consist of 2-input addition and multiplication gates over $\mathbb{F}_p$, whose wire labels lie in the range $[0, \ldots, L]$. Informally, a threshold L-levelled SHE scheme supports homomorphic evaluation of any circuit in the family $\mathcal{C}(L)$ with the provision for distributed (threshold) decryption, where the input wire values v_i are mapped to ciphertexts (at level L) by encrypting $\chi(v_i)$.

As remarked in the introduction we could also, as in [30], extend the circuit family $\mathcal{C}(L)$ to include gates which process N input values at once as

$$N\text{-Add}\left(\langle u_1, \ldots, u_N \rangle, \langle v_1, \ldots, v_N \rangle\right) := \langle u_1 + v_1, \ldots, u_N + v_N \rangle,$$
$$N\text{-Mult}\left(\langle u_1, \ldots, u_N \rangle, \langle v_1, \ldots, v_N \rangle\right) := \langle u_1 \times v_1, \ldots, u_N \times v_N \rangle.$$

But such an optimization of the underlying circuit is orthogonal to our consideration. However, the underlying L-levelled packed SHE scheme supports such operations on its underlying plaintext (we will just not consider these operations in our circuits being evaluated).

We can evaluate subcircuits in $\mathcal{C}(L)$; and this is how we will describe the homomorphic evaluation below (this will later help us to argue the correctness property of our general MPC protocol). In particular if $C \in \mathcal{C}(L)$, we

can deal with sub-circuits C^{sub} of C whose input wires have labels $\mathfrak{l}_1^{in}, \ldots, \mathfrak{l}_{\ell_{in}}^{in}$, and whose output wires have labels $\mathfrak{l}_1^{out}, \ldots, \mathfrak{l}_{\ell_{out}}^{out}$, where $\mathfrak{l}_i^{in}, \mathfrak{l}_i^{out} \in [0, \ldots, L]$. Then given ciphertexts $\mathfrak{c}_1, \ldots, \mathfrak{c}_{\ell_{in}}$ encrypting the messages $\mathbf{m}_1, \ldots, \mathbf{m}_{\ell_{in}}$, for which the ciphertexts are at level $\mathfrak{l}_1^{in}, \ldots, \mathfrak{l}_{\ell_{in}}^{in}$, the homomorphic evaluation function will produce ciphertexts $\hat{\mathfrak{c}}_1, \ldots, \hat{\mathfrak{c}}_{\ell_{out}}$, at levels $\mathfrak{l}_1^{out}, \ldots, \mathfrak{l}_{\ell_{out}}^{out}$, which encrypt the messages corresponding to evaluating C^{sub} on the components of the vectors $\mathbf{m}_1, \ldots, \mathbf{m}_{\ell_{in}}$ in a SIMD manner. More formally:

Definition 2 (Threshold L-levelled Packed SHE). *An L-levelled public key packed somewhat homomorphic encryption (SHE) scheme with the underlying message space $\mathcal{M} = \mathbb{F}_p^N$, public key space $\mathcal{PK}$, secret key space $\mathcal{SK}$, evaluation key space $\mathcal{EK}$, ciphertext space $\mathcal{CT}$ and distributed decryption key space $\mathcal{DK}_i$ for $i \in [1, \ldots, n]$ is a collection of the following PPT algorithms, parametrized by a computational security parameter κ and a statistical security parameter* sec*:*

1. $\mathsf{SHE.KeyGen}(1^\kappa, 1^{\mathsf{sec}}, n, t) \rightarrow (\mathsf{pk}, \mathsf{ek}, \mathsf{sk}, \mathsf{dk}_1, \ldots, \mathsf{dk}_n)$*: The key generation algorithm outputs a public key* $\mathsf{pk} \in \mathcal{PK}$*, a public evaluation key* $\mathsf{ek} \in \mathcal{EK}$*, a secret key* $\mathsf{sk} \in \mathcal{SK}$ *and n keys* $(\mathsf{dk}_1, \ldots, \mathsf{dk}_n)$ *for the distributed decryption, with* $\mathsf{dk}_i \in \mathcal{DK}_i$*.*
2. $\mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{m}, r) \rightarrow (\mathfrak{c}, L)$*: The encryption algorithm computes a ciphertext* $\mathfrak{c} \in \mathcal{CT}$*, which encrypts a plaintext vector* $\mathbf{m} \in \mathcal{M}$ *under the public key* pk *using the randomness*[2] *r and outputs $(\mathfrak{c}, L)$ to indicate that the* associated level *of the ciphertext is L.*
3. $\mathsf{SHE.Dec}_{\mathsf{sk}}(\mathfrak{c}, \mathfrak{l}) \rightarrow \mathbf{m}'$*: The decryption algorithm decrypts a ciphertext* $\mathfrak{c} \in \mathcal{CT}$ *of associated level* $\mathfrak{l}$ *where* $\mathfrak{l} \in [0, \ldots, L]$ *using the decryption key* sk *and outputs a plaintext* $\mathbf{m}' \in \mathcal{M}$*. We say that* $\mathbf{m}'$ *is the* plaintext associated with $\mathfrak{c}$*.*
4. $\mathsf{SHE.ShareDec}_{\mathsf{dk}_i}(\mathfrak{c}, \mathfrak{l}) \rightarrow \bar{\mu}_i$*: The share decryption algorithm takes a ciphertext* $\mathfrak{c}$ *with associated level* $\mathfrak{l} \in [0, \ldots, L]$*, a key* dk_i *for the distributed decryption, and computes a decryption share* $\bar{\mu}_i$ *of* $\mathfrak{c}$*.*
5. $\mathsf{SHE.ShareCombine}((\mathfrak{c}, \mathfrak{l}), \{\bar{\mu}_i\}_{i \in [1,\ldots,n]}) \rightarrow \mathbf{m}'$*: The share combine algorithm takes a ciphertext* $\mathfrak{c}$ *with associated level* $\mathfrak{l} \in [0, \ldots, L]$ *and a set of n decryption shares and outputs a plaintext* $\mathbf{m}' \in \mathcal{M}$*.*
6. $\mathsf{SHE.Eval}_{\mathsf{ek}}(C^{\mathsf{sub}}, (\mathfrak{c}_1, \mathfrak{l}_1^{in}), \ldots, (\mathfrak{c}_{\ell_{in}}, \mathfrak{l}_{\ell_{in}}^{in})) \rightarrow (\hat{\mathfrak{c}}_1, \mathfrak{l}_1^{out}), \ldots, (\hat{\mathfrak{c}}_{\ell_{out}}, \mathfrak{l}_{\ell_{out}}^{out})$*: The homomorphic evaluation algorithm is a deterministic polynomial time algorithm (polynomial in L, ℓ_{in}, ℓ_{out} and κ) that takes as input the evaluation key* ek*, a sub-circuit C^{sub} of a circuit $C \in \mathcal{C}(L)$ with ℓ_{in} input gates and ℓ_{out} output gates as well as a set of ℓ_{in} ciphertexts $\mathfrak{c}_1, \ldots, \mathfrak{c}_{\ell_{in}}$, with associated level $\mathfrak{l}_1^{in}, \ldots, \mathfrak{l}_{\ell_{in}}^{in}$, and outputs ℓ_{out} ciphertexts $\hat{\mathfrak{c}}_1, \ldots, \hat{\mathfrak{c}}_{\ell_{out}}$, with associated levels $\mathfrak{l}_1^{out}, \ldots, \mathfrak{l}_{\ell_{out}}^{out}$ respectively, where each $\mathfrak{l}_i^{in}, \mathfrak{l}_i^{out} \in [0, \ldots, L]$ is the label associated to the given input/output wire in C^{sub}.*
   Algorithm SHE.Eval *associates the input ciphertexts with the input gates of C^{sub} and homomorphically evaluates C^{sub} gate by gate in an SIMD manner on the components of the input messages. For this,* SHE.Eval *consists of separate algorithms* SHE.Add *and* SHE.Mult *for homomorphically evaluating addition and multiplication gates respectively. More specifically, given two ciphertexts $(\mathfrak{c}_1, \mathfrak{l}_1)$ and $(\mathfrak{c}_2, \mathfrak{l}_2)$ with associated levels $\mathfrak{l}_1$ and $\mathfrak{l}_2$ respectively where $\mathfrak{l}_1, \mathfrak{l}_2 \in [0, \ldots, L]$ then*[3]*:*
   - $\mathsf{SHE.Add}_{\mathsf{ek}}((\mathfrak{c}_1, \mathfrak{l}_1), (\mathfrak{c}_2, \mathfrak{l}_2)) \rightarrow (\mathfrak{c}_{\mathsf{Add}}, \min(\mathfrak{l}_1, \mathfrak{l}_2))$*: The deterministic polynomial time addition algorithm takes as input $(\mathfrak{c}_1, \mathfrak{l}_1), (\mathfrak{c}_2, \mathfrak{l}_2)$ and outputs a ciphertext $\mathfrak{c}_{\mathsf{Add}}$ with associated level $\min(\mathfrak{l}_1, \mathfrak{l}_2)$.*
   - $\mathsf{SHE.Mult}_{\mathsf{ek}}((\mathfrak{c}_1, \mathfrak{l}_1), (\mathfrak{c}_2, \mathfrak{l}_2)) \rightarrow (\mathfrak{c}_{\mathsf{Mult}}, \min(\mathfrak{l}_1, \mathfrak{l}_2) - 1)$*: The deterministic polynomial time multiplication algorithm takes as input $(\mathfrak{c}_1, \mathfrak{l}_1), (\mathfrak{c}_2, \mathfrak{l}_2)$ and outputs a ciphertext $\mathfrak{c}_{\mathsf{Mult}}$ with associated level $\min(\mathfrak{l}_1, \mathfrak{l}_2) - 1$.*
   - $\mathsf{SHE.ScalarMult}_{\mathsf{ek}}((\mathfrak{c}_1, \mathfrak{l}_1), \mathbf{a}) \rightarrow (\mathfrak{c}_{\mathsf{Scalar}}, \mathfrak{l}_1)$*: The deterministic polynomial time scalar multiplication algorithm takes as input $(\mathfrak{c}_1, \mathfrak{l}_1)$ and a plaintext $\mathbf{a} \in \mathcal{M}$ and outputs a ciphertext $\mathfrak{c}_{\mathsf{Scalar}}$ with associated level $\mathfrak{l}_1$.*
7. $\mathsf{SHE.Pack}_{\mathsf{ek}}((\mathfrak{c}_1, \mathfrak{l}_1), \ldots, (\mathfrak{c}_N, \mathfrak{l}_N)) \rightarrow (\mathfrak{c}, \min(\mathfrak{l}_1, \ldots, \mathfrak{l}_N))$*: If $\mathfrak{c}_i$ is a ciphertext with associated plaintext $\chi(m_i)$, then this procedure produces a ciphertext $(\mathfrak{c}, \min(\mathfrak{l}_1, \ldots, \mathfrak{l}_N))$ with associated plaintext $\mathbf{m} = (m_1, \ldots, m_N)$.*
8. $\mathsf{SHE.Unpack}_{\mathsf{ek}}(\mathfrak{c}, \mathfrak{l}) \rightarrow ((\mathfrak{c}_1, \mathfrak{l}), \ldots, (\mathfrak{c}_N, \mathfrak{l}))$*: If $\mathfrak{c}$ is a ciphertext with associated plaintext $\mathbf{m} = (m_1, \ldots, m_N)$, then this procedure produces N ciphertexts $(\mathfrak{c}_1, \mathfrak{l}), \ldots, (\mathfrak{c}_N, \mathfrak{l})$ such that $\mathfrak{c}_i$ has associated plaintext $\chi(m_i)$.*

[2] In the paper, unless it is explicitly specified, we assume that some randomness has been used for encryption.

[3] Without loss of generality we assume that we can perform homomorphic operations on ciphertexts of different levels, since we can always deterministically downgrade the ciphertext level of any ciphertext to any value between zero and its current value using $\mathsf{SHE.LowerLevel}_{\mathsf{ek}}$.

9. $\mathsf{SHE.LowerLevel}_{\mathsf{ek}}((\mathfrak{c}, \mathfrak{l}), \mathfrak{l}') \rightarrow (\mathfrak{c}, \mathfrak{l}')$*: This procedure, for* $\mathfrak{l}' < \mathfrak{l}$*, produces a ciphertext with the same associated plaintext as* $(\mathfrak{c}, \mathfrak{l})$*, but at level* $\mathfrak{l}'$*.* □

We require the following homomorphic property to be satisfied:

– *Somewhat Homomorphic SIMD Property*: Let $C^{\mathsf{sub}} : \mathbb{F}_p^{\ell_{in}} \rightarrow \mathbb{F}_p^{\ell_{out}}$ be any sub-circuit of a circuit C in the family $\mathcal{C}(L)$ with respective inputs $\mathbf{m}_1, \ldots, \mathbf{m}_{\ell_{in}} \in \mathcal{M}$, such that C^{sub} when evaluated N times in an SIMD fashion on the N components of the vectors $\mathbf{m}_1, \ldots, \mathbf{m}_{\ell_{in}}$, produces N sets of ℓ_{out} output values $\hat{\mathbf{m}}_1, \ldots, \hat{\mathbf{m}}_{\ell_{in}} \in \mathcal{M}$. Moreover, for $i \in [1, \ldots, \ell_{in}]$ let $\mathfrak{c}_i$ be a ciphertext of level $\mathfrak{l}_i^{in}$ with associated plaintext vector $\mathbf{m}_i$ and let $(\hat{\mathfrak{c}}_1, \mathfrak{l}_1^{out}), \ldots, (\hat{\mathfrak{c}}_{\ell_{out}}, \mathfrak{l}_{\ell_{out}}^{out}) = \mathsf{SHE.Eval}_{\mathsf{ek}}(C^{\mathsf{sub}}, (\mathfrak{c}_1, \mathfrak{l}_1^{in}), \ldots, (\mathfrak{c}_{\ell_{in}}, \mathfrak{l}_{\ell_{in}}^{in}))$. Then the following holds with probability one for each $i \in [1, \ldots, \ell_{out}]$:

$$\mathsf{SHE.Dec}_{\mathsf{sk}}(\hat{\mathfrak{c}}_i, \mathfrak{l}_i^{out}) = \hat{\mathbf{m}}_i.$$

We also require the following security properties:

– *Key Generation Security*: Let S and D_i be the random variables which denote the probability distribution with which the secret key sk and the ith key dk_i for the distributed decryption is selected from $\mathcal{SK}$ and $\mathcal{DK}_i$ by $\mathsf{SHE.KeyGen}$ for $i = 1, \ldots, n$. Moreover, for a set $I \subseteq \{1, \ldots, n\}$, let D_I denote the random variable which denote the probability distribution with which the set of keys for the distributed decryption, belonging to the indices in I, are selected from the corresponding $\mathcal{DK}_i$s by $\mathsf{SHE.KeyGen}$. Then the following two properties hold:
  - *Correctness*: For any set $I \subseteq \{1, \ldots, n\}$ with $|I| \geq t + 1$, $H(S|D_I) = 0$. Here $H(X|Y)$ denotes the conditional entropy of a random variable X with respect to a random variable Y [16].
  - *Privacy*: For any set $I \subset \{1, \ldots, n\}$ with $|I| \leq t$, $H(S|D_I) = H(S)$.
– *Semantic Security*: For every set $I \subset \{1, \ldots, n\}$ with $|I| \leq t$ and all PPT adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ in the following game is negligible in κ:
  - *Key Generation*: The challenger runs $\mathsf{SHE.KeyGen}(1^\kappa, 1^{\mathsf{sec}}, n, t)$ to obtain $(\mathsf{pk}, \mathsf{ek}, \mathsf{sk}, \mathsf{dk}_1, \ldots, \mathsf{dk}_n)$ and sends pk, ek and $\{\mathsf{dk}_i\}_{i \in I}$ to $\mathcal{A}$.
  - *Challenge*: $\mathcal{A}$ sends plaintexts $\mathbf{m}_0, \mathbf{m}_1 \in \mathcal{M}$ to the challenger, who randomly selects $b \in \{0, 1\}$ and sends $(\mathfrak{c}, L) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{m}_b, r)$ for some randomness r to $\mathcal{A}$.
  - *Output*: $\mathcal{A}$ outputs b'.

  The advantage of $\mathcal{A}$ in the above game is defined to be $|\frac{1}{2} - \Pr[b' = b]|$.
– *Correct Share Decryption*: For any $(\mathsf{pk}, \mathsf{ek}, \mathsf{sk}, \mathsf{dk}_1, \ldots, \mathsf{dk}_n)$ obtained as the output of $\mathsf{SHE.KeyGen}$, the following should hold for any ciphertext $(\mathfrak{c}, \mathfrak{l})$ with associated level $\mathfrak{l} \in [0, \ldots, L]$:

$$\mathsf{SHE.Dec}_{\mathsf{sk}}(\mathfrak{c}, \mathfrak{l}) = \mathsf{SHE.ShareCombine}((\mathfrak{c}, \mathfrak{l}), \{\mathsf{SHE.ShareDec}_{\mathsf{dk}_i}(\mathfrak{c}, \mathfrak{l})\}_{i \in [1, \ldots, n]}).$$

– *Share Simulation Indistinguishability*: There exists a PPT simulator $\mathsf{SHE.ShareSim}$, which on input a subset $I \subset \{1, \ldots, n\}$ of size at most t, a ciphertext $(\mathfrak{c}, \mathfrak{l})$ of level $\mathfrak{l} \in [0, \ldots, L]$, a plaintext $\mathbf{m}$ and $|I|$ decryption shares $\{\bar{\mu}_i\}_{i \in I}$ outputs $n - |I|$ simulated decryption shares $\{\bar{\mu}_j^*\}_{j \in \overline{I}}$ with the following property: For any $(\mathsf{pk}, \mathsf{ek}, \mathsf{sk}, \mathsf{dk}_1, \ldots, \mathsf{dk}_n)$ obtained as the output of $\mathsf{SHE.KeyGen}$, any subset $I \subset \{1, \ldots, n\}$ of size at most t, any $\mathbf{m} \in \mathcal{M}$ and any $(\mathfrak{c}, \mathfrak{l})$ where $\mathbf{m} = \mathsf{SHE.Dec}_{\mathsf{sk}}(\mathfrak{c}, \mathfrak{l})$, the following distributions are statistically indistinguishable:

$$(\{\bar{\mu}_i\}_{i \in I}, \mathsf{SHE.ShareSim}((\mathfrak{c}, \mathfrak{l}), \mathbf{m}, \{\bar{\mu}_i\}_{i \in I})) \overset{s}{\approx} \left(\{\bar{\mu}_i\}_{i \in I}, \{\bar{\mu}_j\}_{j \in \overline{I}}\right),$$

  where for all $i \in [1, \ldots, n]$, $\bar{\mu}_i = \mathsf{SHE.ShareDec}_{\mathsf{dk}_i}(\mathfrak{c}, \mathfrak{l})$. We require in particular that the statistical distance between the two distributions is bounded by $2^{-\mathsf{sec}}$. Moreover

$$\mathsf{SHE.ShareCombine}((\mathfrak{c}, \mathfrak{l}), \{\bar{\mu}_i\}_{i \in I} \cup \mathsf{SHE.ShareSim}((\mathfrak{c}, \mathfrak{l}), \mathbf{m}, \{\bar{\mu}_i\}_{i \in I}))$$

  outputs the result $\mathbf{m}$. Here $\overline{I}$ denotes the complement of the set I; i.e. $\overline{I} = \{1, \ldots, n\} \setminus I$.

In Section 7 we instantiate the abstract syntax with a threshold SHE scheme based on the BGV scheme [10]. We pause to note the difference between our underlying SHE, which is just an SHE scheme which supports distributed decryption, and that of [1] which requires a special key homomorphic FHE scheme.

4 MPC from SHE – The Semi-honest Settings

In this section we present our generic MPC protocol for the computation of any arbitrary depth d circuit using an abstract threshold L-levelled SHE scheme. For the ease of exposition we first concentrate on the case of semi-honest security, and then we deal with active security in Section 5.

Functionality $\mathcal{F}_f$

$\mathcal{F}_f$ interacts with the parties $P_1, \ldots, P_n$ and the adversary $\mathcal{S}$ and is parametrized by an n-input function $f : \mathbb{F}_p^n \to \mathbb{F}_p$.

- Upon receiving (sid, i, x_i) from the party P_i for every $i \in [1, \ldots, n]$ where $x_i \in \mathbb{F}_p$, compute $y = C(x_1, \ldots, x_n)$, send (sid, y) to all the parties and the adversary $\mathcal{S}$ and halt. Here C denotes the (publicly known) well formed arithmetic circuit over $\mathbb{F}_p$ representing the function f.

Fig. 1. The Ideal Functionality for Computing a Given Function

Without loss of generality we make the simplifying assumption that the function f to be computed takes a single input from each party and has a single output; specifically $f : \mathbb{F}_p^n \to \mathbb{F}_p$. The ideal functionality $\mathcal{F}_f$ presented in Figure 1 computes such a given function f, represented by a well formed circuit C. We will present a protocol to realise the ideal functionality $\mathcal{F}_f$ in a hybrid model in which we are given access to an ideal functionality $\mathcal{F}_{\text{SETUPGEN}}$ which implements a distributed key generation for the underlying SHE scheme. In particular the $\mathcal{F}_{\text{SETUPGEN}}$ functionality presented in Figure 2 computes the public key, secret key, evaluation key and the keys for the distributed decryption of an L-levelled SHE scheme, distributes the public key and the evaluation key to all the parties and sends the ith key dk_i (for the distributed decryption) to the party P_i for each $i \in [1, \ldots, n]$. In addition, the functionality also computes a random encryption $\mathfrak{c}_{\underline{\mathbf{1}}}$ with associated plaintext $\underline{\mathbf{1}} = (1, \ldots, 1) \in \mathcal{M}$ and sends it to all the parties. Looking ahead, $\mathfrak{c}_{\underline{\mathbf{1}}}$ will be required while proving the security of our MPC protocol. The ciphertext $\mathfrak{c}_{\underline{\mathbf{1}}}$ is at level one, as we only need it to pre-multiply the ciphertexts which are going to be decrypted via the distributed decryption protocol; thus the output of a multiplication by $\mathfrak{c}_{\underline{\mathbf{1}}}$ need only be at level zero. Looking ahead, this ensures that (with respect to our instantiation of SHE) the noise is kept to a minimum at this stage of the protocol.

Functionality $\mathcal{F}_{\text{SETUPGEN}}$

$\mathcal{F}_{\text{SETUPGEN}}$ interacts with the parties $P_1, \ldots, P_n$ and the adversary $\mathcal{S}$ and is parametrized by an L-levelled SHE scheme.

- Upon receiving (sid, i) from the party P_i for every $i \in [1, \ldots, n]$, compute $(\mathsf{pk}, \mathsf{ek}, \mathsf{sk}, \mathsf{dk}_1, \ldots, \mathsf{dk}_n) = \mathsf{SHE.KeyGen}(1^\kappa, 1^{\mathsf{sec}}, n, t)$ and $(\mathfrak{c}_{\underline{\mathbf{1}}}, 1) = \mathsf{SHE.LowerLevel}_{\mathsf{ek}}((\mathsf{SHE.Enc}_{\mathsf{pk}}(\underline{\mathbf{1}}, r), 1)$ for $\underline{\mathbf{1}} = (1, \ldots, 1) \in \mathcal{M}$ and some randomness r. Finally send $(\mathsf{sid}, \mathsf{pk}, \mathsf{ek}, \mathsf{dk}_i, (\mathfrak{c}_{\underline{\mathbf{1}}}, 1))$ to the party P_i for every $i \in [1, \ldots, n]$ and halt.

Fig. 2. The Ideal Functionality for Key Generation

4.1 The MPC Protocol in the $\mathcal{F}_{\text{SETUPGEN}}$-hybrid Model

Here we present our MPC protocol Π_f^{SH} in the $\mathcal{F}_{\text{SETUPGEN}}$-hybrid model. Let C be the (well formed) arithmetic circuit representing the function f and C^{aug} be the associated augmented circuit (which includes the necessary Refresh gates). The protocol Π_f^{SH} (see Figure 3) runs in two phases: offline and online. The computation performed in the offline phase is completely independent of the circuit and (private) inputs of the parties and therefore can be carried out well ahead of the time (namely the online phase) when the function and inputs are known. If the parties have more than one

input/output then one can apply packing/unpacking at the input/output stages of the protocol; we leave this minor modification to the reader.

In the offline phase, the parties interact with $\mathcal{F}_{\text{SETUPGEN}}$ to obtain the public key, evaluation key and their respective keys for performing distributed decryption, corresponding to a threshold L-levelled SHE scheme. Next each party sends encryptions of ζ random elements and then additively combines them (by applying the homomorphic addition to the ciphertexts encrypting the random elements) to generate ζ ciphertexts at level L of truly random elements (unknown to the adversary). Here ζ is assumed to be large enough, so that for a typical circuit it is more than the number of refresh gates in the circuit, i.e. $\zeta > \mathbb{G}_R$. Looking ahead, these random ciphertexts created in the offline phase are used in the online phase to evaluate refresh gates by (homomorphically) masking the messages associated with the input wires of a refresh gate.

During the online phase, the parties encrypt their private inputs and distribute the corresponding ciphertexts to all other parties. These ciphertexts are transmitted at level one, thus consuming low bandwidth, and are then elevated to level L by the use of a following Refresh gate (which would have been inserted by the circuit augmentation process). Note that the inputs of the parties are in $\mathbb{F}_p$ and so the parties first apply the mapping χ (embedding $\mathbb{F}_p$ into the message space $\mathcal{M}$ of SHE) before encrypting their private inputs.

The input stage is followed by the homomorphic evaluation of C^{aug} as follows: The addition and multiplication gates are evaluated locally using the addition and multiplication algorithm of the SHE. For each refresh gate, the parties execute the following protocol to enable a "multiparty bootstrapping" of the input ciphertexts: the parties pick one of the random ciphertext created in the offline phase (for each refresh gate a different ciphertext is used) and perform the following computation to refresh N ciphertexts with levels in the range $[1, \ldots, L]$ and obtain N fresh level L ciphertexts, with the associated messages unperturbed:

- Let $(\mathfrak{c}_1, \mathfrak{l}_1), \ldots, (\mathfrak{c}_N, \mathfrak{l}_N)$ be the N ciphertexts with associated plaintexts $\chi(z_1), \ldots, \chi(z_N)$ with every $z_i \in \mathbb{F}_p$, that need to be refreshed (i.e. they are the inputs of a refresh gate).
- The N ciphertexts are then (locally) packed into a single ciphertext $\mathfrak{c}$, which is then homomorphically masked with a random ciphertext from the offline phase.
- The resulting masked ciphertext is then publicly opened via distributed decryption, This allows for the creation of a fresh encryption of the opened value at level L.
- The resulting fresh encryption is then homomorphically unmasked so that its associated plaintext is the same as original plaintext prior to the original masking.
- This fresh (unmasked) ciphertext is then unpacked to obtain N fresh ciphertexts, having the same associated plaintexts as the original N ciphertexts $\mathfrak{c}_i$ but at level L.

By packing the ciphertexts together we only need to invoke distributed decryption once, instead of N times. This leads to a more communication efficient online phase, since the distributed decryption is the only operation that demands communication. Without affecting the correctness of the above technique, but to ensure security, we add an additional step while doing the masking: the parties homomorphically pre-multiply the ciphertext $\mathfrak{c}$ with $\mathfrak{c}_{\underline{\mathbf{1}}}$ before masking. Recall that $\mathfrak{c}_{\underline{\mathbf{1}}}$ is an encryption of $\underline{\mathbf{1}} \in \mathcal{M}$ generated by $\mathcal{F}_{\text{SETUPGEN}}$ and so by doing the above operation, the plaintext associated with $\mathfrak{c}$ remains the same. During the simulation in the security proof, this step allows the simulator to set the decrypted value to the random mask (irrespective of the circuit inputs), by playing the role of $\mathcal{F}_{\text{SETUPGEN}}$ and replacing $\mathfrak{c}_{\underline{\mathbf{1}}}$ with $\mathfrak{c}_{\underline{\mathbf{0}}}$, a random encryption of $\underline{\mathbf{0}} = (0, \ldots, 0)$. Furthermore, this step explains the reason why we made provision for an extra multiplication during circuit augmentation by insisting that the refresh gates take inputs with labels in $[1, \ldots, L]$, instead of $[0, \ldots, L]$; the details are available in the simulation proof of security of our MPC protocol.

Finally, the function output y is obtained by another distributed decryption of the output ciphertext. However, this step is also not secure unless the ciphertext is randomized again by pre-multiplication by $\mathfrak{c}_{\underline{\mathbf{1}}}$ and adding n encryptions of $\underline{\mathbf{0}}$ where each party contributes one encryption. In the simulation, the simulator gives encryption of $\chi(y)$ on behalf of one honest party and replaces $\mathfrak{c}_{\underline{\mathbf{1}}}$ by $\mathfrak{c}_{\underline{\mathbf{0}}}$, letting the output ciphertext correspond to the actual output y, even though the circuit is evaluated with zero as the inputs of the honest parties during the simulation (the simulator will not know the real inputs of the honest parties and thus will simulate them with zero). A similar idea was also used in [23]; details can be found in the security proof.

Intuitively, privacy follows because at any stage of the computation, the keys of the honest parties for the distributed decryption are not revealed and so the adversary will not be able to decrypt any intermediate ciphertext. Correctness

Protocol Π_f^{SH}

Let C^{aug} denote an augmented circuit for a well formed circuit C over $\mathbb{F}_p$ representing f and let SHE be a threshold L-levelled SHE. Moreover, let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be the set of n parties For the session ID sid the parties do the following:

Offline Computation: Every party $P_i \in \mathcal{P}$ does the following:

- Call $\mathcal{F}_{\text{SETUPGEN}}$ with (sid, i) and receive $(\mathsf{sid}, \mathsf{pk}, \mathsf{ek}, \mathsf{dk}_i, (\mathfrak{c}_{\underline{\mathbf{1}}}, 1))$.
- Randomly select ζ plaintexts $\mathbf{m}_{i,1}, \ldots, \mathbf{m}_{i,\zeta} \in \mathcal{M}$, and compute $(\mathfrak{c}_{\mathbf{m}_{i,k}}, L) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{m}_{i,k}, r_{i,k})$. Send $(\mathsf{sid}, i, (\mathfrak{c}_{\mathbf{m}_{i,1}}, L), \ldots, (\mathfrak{c}_{\mathbf{m}_{i,\zeta}}, L))$ to all parties in $\mathcal{P}$.
- Upon receiving $(\mathsf{sid}, j, (\mathfrak{c}_{\mathbf{m}_{j,1}}, L), \ldots, (\mathfrak{c}_{\mathbf{m}_{j,\zeta}}, L))$ from all parties $P_j \in \mathcal{P}$, apply $\mathsf{SHE.Add}$ for $1 \leq k \leq \zeta$, on $(\mathfrak{c}_{\mathbf{m}_{1,k}}, L), \ldots, (\mathfrak{c}_{\mathbf{m}_{n,k}}, L)$, set the resultant ciphertext as the kth offline ciphertext $\mathfrak{c}_{\mathbf{m}_k}$ with the (unknown) associated plaintext $\mathbf{m}_k = \mathbf{m}_{1,k} + \cdots + \mathbf{m}_{n,k}$.

Online Computation: Every party $P_i \in \mathcal{P}$ does the following:

- **Input Stage**: On having input $x_i \in \mathbb{F}_p$, compute $(\mathfrak{c}_{\mathbf{x}_i}, 1) = \mathsf{SHE.LowerLevel}_{\mathsf{ek}}(\mathsf{SHE.Enc}_{\mathsf{pk}}(\chi(x_i), r_i), 1)$ with randomness r_i and send $(\mathsf{sid}, i, (\mathfrak{c}_{\mathbf{x}_i}, 1))$ to each party. Receive $(\mathsf{sid}, j, (\mathfrak{c}_{\mathbf{x}_j}, 1))$ from each party $P_j \in \mathcal{P}$.
- **Computation Stage**: Associate the ciphertexts received with the corresponding input wires of C^{aug} and then homomorphically evaluate the circuit C^{aug} gate by gate as follows:
  - **Addition Gate and Multiplication Gate**: Given $(\mathfrak{c}_1, \mathfrak{l}_1)$ and $(\mathfrak{c}_2, \mathfrak{l}_2)$ associated with the input wires of the gate where $\mathfrak{l}_1, \mathfrak{l}_2 \in [1, \ldots, L]$, locally compute $(\mathfrak{c}, \mathfrak{l}) = \mathsf{SHE.Add}_{\mathsf{ek}}((\mathfrak{c}_1, \mathfrak{l}_1), (\mathfrak{c}_2, \mathfrak{l}_2))$ with $\mathfrak{l} = \min(\mathfrak{l}_1, \mathfrak{l}_2)$ for an addition gate and $(\mathfrak{c}, \mathfrak{l}) = \mathsf{SHE.Mult}_{\mathsf{ek}}((\mathfrak{c}_1, \mathfrak{l}_1), (\mathfrak{c}_2, \mathfrak{l}_2))$ with $\mathfrak{l} = \min(\mathfrak{l}_1, \mathfrak{l}_2) - 1$ for a multiplication gate; for the multiplication gate, $\mathfrak{l}_1, \mathfrak{l}_2 \in [2, \ldots, L]$, instead of $[1, \ldots, L]$. Associate $(\mathfrak{c}, \mathfrak{l})$ with the output wire of the gate.
  - **Refresh Gate**: For the kth refresh gate in the circuit, the kth offline ciphertext $(\mathfrak{c}_{\mathbf{m}_k}, L)$ is used. Let $(\mathfrak{c}_1, \mathfrak{l}_1), \ldots, (\mathfrak{c}_N, \mathfrak{l}_N)$ be the ciphertexts associated with the input wires of the refresh gate where $\mathfrak{l}_1, \ldots, \mathfrak{l}_N \in [1, \ldots, L]$:
    - **Packing**: Locally compute $(\mathfrak{c}_{\mathbf{z}}, \mathfrak{l}) = \mathsf{SHE.Pack}_{\mathsf{ek}}(\{(\mathfrak{c}_i, \mathfrak{l}_i)\}_{i \in [1,\ldots,N]})$ where $\mathfrak{l} = \min(\mathfrak{l}_1, \ldots, \mathfrak{l}_N)$.
    - **Masking**: Locally compute $(\mathfrak{c}_{\mathbf{z}+\mathbf{m}_k}, 0) = \mathsf{SHE.Add}_{\mathsf{ek}}(\mathsf{SHE.Mult}_{\mathsf{ek}}((\mathfrak{c}_{\mathbf{z}}, \mathfrak{l}), (\mathfrak{c}_{\underline{\mathbf{1}}}, 1)), (\mathfrak{c}_{\mathbf{m}_k}, L))$
    - **Decrypting**: Locally compute the decryption share $\bar{\mu}_i = \mathsf{SHE.ShareDec}_{\mathsf{dk}_i}(\mathfrak{c}_{\mathbf{z}+\mathbf{m}_k}, 0)$ and send $(\mathsf{sid}, i, \bar{\mu}_i)$ to every other party. On receiving $(\mathsf{sid}, j, \bar{\mu}_j)$ from every $P_j \in \mathcal{P}$, compute the plaintext $\mathbf{z} + \mathbf{m}_k = \mathsf{SHE.ShareCombine}((\mathfrak{c}_{\mathbf{z}+\mathbf{m}_k}, 0), \{\bar{\mu}_j\}_{j \in [1,\ldots,n]})$.
    - **Re-encrypting**: Locally re-encrypt $\mathbf{z} + \mathbf{m}_k$ by computing $(\hat{\mathfrak{c}}_{\mathbf{z}+\mathbf{m}_k}, L) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{z} + \mathbf{m}_k, r)$ using a publicly known (common) randomness r, (This can simply be the zero string for our BGV instantiation, we only need to map the known plaintext into a ciphertext element).
    - **Unmasking**: Locally subtract $(\mathfrak{c}_{\mathbf{m}_k}, L)$ from $(\hat{\mathfrak{c}}_{\mathbf{z}+\mathbf{m}_k}, L)$ to obtain $(\hat{\mathfrak{c}}_{\mathbf{z}}, L)$.
    - **Unpacking**: Locally compute $(\hat{\mathfrak{c}}_1, L), \ldots, (\hat{\mathfrak{c}}_N, L) = \mathsf{SHE.Unpack}_{\mathsf{ek}}(\hat{\mathfrak{c}}_{\mathbf{z}}, L)$ and associate $(\hat{\mathfrak{c}}_1, L), \ldots, (\hat{\mathfrak{c}}_N, L)$ with the output wires of the refresh gate.
- **Output Stage**: Let $(\mathfrak{c}, \mathfrak{l})$ be the ciphertext associated with the output wire of C^{aug} where $\mathfrak{l} \in [1, \ldots, L]$.
  - **Randomization:** Compute a random encryption $(\mathfrak{c}_i, L) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\underline{\mathbf{0}}, r_i')$ of $\underline{\mathbf{0}} = (0, \ldots, 0)$ and send $(\mathsf{sid}, i, (\mathfrak{c}_i, L))$ to every other party. On receiving $(\mathsf{sid}, j, (\mathfrak{c}_j, L))$ from every $P_j \in \mathcal{P}$, apply $\mathsf{SHE.Add}$ on $\{(\mathfrak{c}_j, L)\}_{j \in [1,\ldots,n]}$ to obtain $(\mathfrak{c}_{\underline{\mathbf{0}}}, L)$. Compute $(\hat{\mathfrak{c}}, 0) = \mathsf{SHE.Add}_{\mathsf{ek}}(\mathsf{SHE.Mult}_{\mathsf{ek}}((\mathfrak{c}, \mathfrak{l}), (\mathfrak{c}_{\underline{\mathbf{1}}}, 1)), (\mathfrak{c}_{\underline{\mathbf{0}}}, L))$.
  - **Output Decryption:** Compute $\bar{\gamma}_i = \mathsf{SHE.ShareDec}_{\mathsf{dk}_i}(\hat{\mathfrak{c}}, 0)$ and send $(\mathsf{sid}, i, \bar{\gamma}_i)$ to every party. On receiving $(\mathsf{sid}, j, \bar{\gamma}_j)$ from every $P_j \in \mathcal{P}$, compute $\mathbf{y} = \mathsf{SHE.ShareCombine}((\hat{\mathfrak{c}}, 0), \{\bar{\gamma}_j\}_{j \in [1,\ldots,n]})$, output y and halt, where $y = \chi^{-1}(\mathbf{y})$.

Fig. 3. The Protocol for Realizing $\mathcal{F}_f$ against a Semi-Honest Adversary in the $\mathcal{F}_{\text{SETUPGEN}}$-hybrid Model

follows from the properties of the SHE and the fact that the level of each ciphertext in the protocol remains in the range $[1, \ldots, L]$, thanks to the refresh gates. So even though the circuit C may have any arbitrary depth $d > L$, we can homomorphically evaluate C using an L-levelled SHE.

We work in the standard Universal Composability (UC) framework of Canetti [13], with static corruption. The UC framework introduces a PPT environment $\mathcal{Z}$ that is invoked on the security parameter 1^κ and an auxiliary input z and oversees the execution of a protocol in one of the two worlds. The "ideal" world execution involves dummy parties $P_1, \ldots, P_n$, an ideal adversary $\mathcal{S}$ who may corrupt some of the dummy parties, and a functionality $\mathcal{F}$. The "real" world execution involves the PPT parties $P_1, \ldots, P_n$ and a real world adversary $\mathcal{A}$ who may corrupt some of the parties. In either of these two worlds, a PPT adversary can corrupt t parties out of the n parties. The environment $\mathcal{Z}$ chooses the input of the parties and may interact with the ideal/real adversary during the execution. At the end of the execution, it has to decide upon and output whether a real or an ideal world execution has taken place.

We let $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\kappa, z)$ denote the random variable describing the output of the environment $\mathcal{Z}$ after interacting with the ideal execution with adversary $\mathcal{S}$, the functionality $\mathcal{F}$, on the security parameter 1^κ and z. Let $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote the ensemble $\{\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$. Similarly let $\text{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(1^\kappa, z)$ denote the random variable describing the output of the environment $\mathcal{Z}$ after interacting in a real execution of a protocol Π with adversary $\mathcal{A}$, the parties $\mathcal{P}$, on the security parameter 1^κ and z. Let $\text{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}$ denote the ensemble $\{\text{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(1^\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$.

Definition 3. *For $n \in \mathbb{N}$, let $\mathcal{F}$ be an n-ary functionality and let Π be an n-party protocol. We say that Π securely realizes $\mathcal{F}$ if for every PPT real world adversary $\mathcal{A}$, there exists a PPT ideal world adversary $\mathcal{S}$, corrupting the same parties, such that the following two distributions are computationally indistinguishable:*

$$\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \overset{c}{\approx} \text{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}.$$

We consider the above definition where it quantifies over different adversaries: passive or active, that corrupts only certain number of parties.

Theorem 1. *Let $f : \mathbb{F}_p^n \to \mathbb{F}_p$ be a function over $\mathbb{F}_p$ represented by a well formed arithmetic circuit C of depth d over $\mathbb{F}_p$. Let $\mathcal{F}_f$ (presented in Figure 1) be the ideal functionality computing f and let* SHE *be a threshold L-levelled SHE scheme. Then the protocol Π_f^{SH} UC-secure realizes $\mathcal{F}_f$ against a static, semi-honest adversary $\mathcal{A}$, corrupting upto $t < n$ parties in the $\mathcal{F}_{\text{SETUPGEN}}$-hybrid Model.*

Proof. We prove the theorem with respect to a generic L-levelled SHE scheme and first consider the correctness. Suppose in the protocol party P_i has input $x_i \in \mathbb{F}_p$. Then we claim the following invariant to hold for each wire w of the circuit C^{aug} during the execution of the protocol: if $(\mathfrak{c}, \mathfrak{l})$ is the ciphertext associated with w during the execution of the protocol where level $\mathfrak{l} \in [1, \ldots, L]$, then $\mathsf{SHE.Dec}_{\mathsf{sk}}(\mathfrak{c}, \mathfrak{l}) = \chi(z)$, where $z \in \mathbb{F}_p$ is the value that would have been associated with w during the evaluation of C^{aug} with input $\boldsymbol{x} = (x_1, \ldots, x_n)$. Before proving the claim, we first recall that due to the introduction of the Refresh gates in C^{aug} and the way circuit is evaluated, every wire in the circuit C^{aug} has label in the range $[1, \ldots, L]$ and the corresponding ciphertext associated with the wire (during the protocol execution) has level in the range $[1, \ldots, L]$. In addition the level of the ciphertext associated to a wire is equal to the label of the wire.

Our invariant is clearly true for the input wires. Assuming that the evaluation of the refresh gates is correct, the invariant is also true for the output of the Refresh gates. That the invariant holds for the rest of the circuit follows from the homomorphic property of the SHE scheme. Finally, the correctness of the refresh gate evaluation follows from the correctness of SHE.Pack, SHE.Unpack, the homomorphic of the underlying SHE; and the fact that all the ciphertexts that are used in evaluating a refresh gate have levels in the range $[0, \ldots, L]$.

We next prove the security. Let $\mathcal{A}$ be a real-world semi-honest adversary corrupting $t < n$ parties and let $T \subset \mathcal{P}$ denote the set of corrupted parties. We now present an ideal-world adversary (simulator) $\mathcal{S}_f^{\text{SH}}$ for $\mathcal{A}$ in Figure 4. The high level idea for the simulator is the following: the simulator takes the input $\{x_i\}_{P_i\in T}$ and interacts with $\mathcal{F}_f$ to obtain the function output y. The simulator then invokes $\mathcal{A}$ with the inputs $\{x_i\}_{P_i\in T}$ and simulates each message that $\mathcal{A}$ would have received in the protocol Π_f^{SH} from the honest parties and from the functionality $\mathcal{F}_{\text{SETUPGEN}}$, stage by stage.

Simulator $\mathcal{S}_f^{\mathsf{SH}}$

Let SHE be an L-levelled SHE scheme. The simulator plays the role of the honest parties and simulates each step of the protocol Π_f^{SH} as follows. The communication of the $\mathcal{Z}$ with the adversary $\mathcal{A}$ is handled as follows: Every input value received by the simulator from $\mathcal{Z}$ is written on $\mathcal{A}$'s input tape. Likewise, every output value written by $\mathcal{A}$ on its output tape is copied to the simulator's output tape (to be read by the environment $\mathcal{Z}$). The simulator then does the following for the session ID sid:

Offline Computation:

- On receiving the message (sid, i) to $\mathcal{F}_{\text{SETUPGEN}}$ from $\mathcal{A}$ for each $P_i \in T$, the simulator invokes $(\mathsf{pk}, \mathsf{ek}, \mathsf{sk}, \mathsf{dk}_1, \ldots, \mathsf{dk}_n) = \mathsf{SHE.KeyGen}(1^\kappa, 1^{\mathsf{sec}}, n, t)$, computes $(\mathfrak{c}_{\underline{\mathbf{0}}}, 1) = \mathsf{SHE.LowerLevel}_{\mathsf{ek}}(\mathsf{SHE.Enc}_{\mathsf{pk}}(\underline{\mathbf{0}}, \cdot), 1)$, and on the behalf of $\mathcal{F}_{\text{SETUPGEN}}$ sends $(\mathsf{sid}, \mathsf{pk}, \mathsf{ek}, \{\mathsf{dk}_i\}_{P_i \in T}, (\mathfrak{c}_{\underline{\mathbf{0}}}, 1))$ to $\mathcal{A}$.
- For each $P_j \notin T$, the simulator computes $(\mathfrak{c}_{\mathbf{m}_{jk}}, L) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{m}_{j,k}, \cdot)$ for $k \in [1, \ldots, \zeta]$ for a randomly chosen $\mathbf{m}_{j,k} \in \mathcal{M}$ and sends $(\mathsf{sid}, j, (\mathfrak{c}_{\mathbf{m}_{j,1}}, L), \ldots, (\mathfrak{c}_{\mathbf{m}_{j,\zeta}}, L))$ to $\mathcal{A}$ on the behalf of the honest parties.
- On receiving $(\mathsf{sid}, i, (\mathfrak{c}_{\mathbf{m}_{i,1}}, L), \ldots, (\mathfrak{c}_{\mathbf{m}_{i,\zeta}}, L))$ from $\mathcal{A}$ for every $P_i \in T$, the simulator decrypts the ciphertexts to get their associated plaintexts $\mathbf{m}_{i,1}, \ldots, \mathbf{m}_{i,\zeta}$; i.e. $\mathbf{m}_{i,k} = \mathsf{SHE.Dec}_{\mathsf{sk}}(\mathfrak{c}_{\mathbf{m}_{i,k}}, L)$. The simulator then applies $\mathsf{SHE.Add}$ on $(\mathfrak{c}_{\mathbf{m}_{1,k}}, L), \ldots, (\mathfrak{c}_{\mathbf{m}_{n,k}}, L)$ and sets the resultant ciphertext as the kth offline ciphertext. Furthermore it sets $\mathbf{m}_k = \mathbf{m}_{1,k} + \ldots + \mathbf{m}_{n,k}$ as the kth offline plaintext.

Online Computation:

- **Input Stage**: For every party $P_j \in \mathcal{P} \setminus T$, the simulator computes a random encryption $(\mathfrak{c}_{\mathbf{x}_j}, 1) = \mathsf{SHE.LowerLevel}_{\mathsf{ek}}(\mathsf{SHE.Enc}_{\mathsf{pk}}(\chi(0), \cdot), 1)$ and sends $(\mathsf{sid}, j, (\mathfrak{c}_{\mathbf{x}_j}, 1))$ to $\mathcal{A}$ on the behalf of every $P_j \in \mathcal{P} \setminus T$. The simulator receives $(\mathsf{sid}, i, (\mathfrak{c}_{\mathbf{x}_i}, 1))$ from $\mathcal{A}$ and obtains the associated plaintext $\mathbf{x}_i$. On the behalf of the parties $P_i \in T$, the simulator sends (sid, i, x_i) to the functionality $\mathcal{F}_f$ and receives y, where $x_i = \chi^{-1}(\mathbf{x}_i) \in \mathbb{F}_p$,
- **Computation Stage**: The simulator performs the local computation (required for the addition, multiplication and refresh gates) as specified in the protocol in order to be synchronized with the adversary with respect to the ciphertexts associated with the wires in the circuit. For the refresh gates, the simulator simulates to $\mathcal{A}$ the communication from the honest parties as follows:
  - **Refresh Gate**: Let this be the kth refresh gate and let $(\mathfrak{c}_{\mathbf{m}_k}, L)$ be the kth offline ciphertext with the associated plaintext $\mathbf{m}_k$, which are known to the simulator while simulating the offline computation. Let $(\mathfrak{c}, 0)$ be the ciphertext obtained after the masking operation. Since $\mathfrak{c}_{\underline{\mathbf{1}}}$ is replaced by $\mathfrak{c}_{\underline{\mathbf{0}}}$ in the simulation, $\mathfrak{c}$ is associated with message $\mathbf{m}_k$. For each $P_i \in T$, on receiving $(\mathsf{sid}, i, \bar{\mu}_i)$ from $\mathcal{A}$ as the decryption shares of $(\mathfrak{c}, 0)$, the simulator computes the simulated decryption shares $\{\bar{\mu}_j^*\}_{P_j \notin T} = \mathsf{SHE.ShareSim}((\mathfrak{c}, 0), \mathbf{m}_k, \{\bar{\mu}_i\}_{P_i \in T})$. The simulator then sends the simulated shares $\{\bar{\mu}_j^*\}_{P_j \notin T}$ to $\mathcal{A}$ as the decryption shares on the behalf of the honest parties.
- **Output Stage**:
  - **Randomization**: On receiving $(\mathsf{sid}, i, (\mathfrak{c}_i, L))$ for every $P_i \in T$ from $\mathcal{A}$, the simulator computes encryptions of $\chi(0)$ for every honest party, except for one honest party, say P_h, it encrypts $\chi(y)$. The simulator sends these ciphertexts to $\mathcal{A}$ on the behalf of the honest parties and then follows the protocol steps to obtain $(\hat{\mathfrak{c}}, 0)$ corresponding to the output wire. Note that the plaintext associated with $\hat{\mathfrak{c}}$ is $\chi(y)$, since $\mathfrak{c}_{\underline{\mathbf{1}}}$ is replaced by $\mathfrak{c}_{\underline{\mathbf{0}}}$ in the simulation and one of the ciphertexts on the behalf of an honest party (for randomization) encrypts $\chi(y)$.
  - On receiving the decryption share $(\mathsf{sid}, i, \bar{\gamma}_i)$ for every $P_i \in T$ from $\mathcal{A}$, the simulator computes the simulated decryption shares $\{\bar{\gamma}_j^*\}_{P_j \in \mathcal{P} \setminus T} = \mathsf{SHE.ShareSim}((\hat{\mathfrak{c}}, 0), \chi(y), \{\bar{\gamma}_i\}_{P_i \in T})$ for the the honest parties $P_j \in \mathcal{P} \setminus T$ and sends $(\mathsf{sid}, j, \bar{\gamma}_j^*)$ as the decryption shares to $\mathcal{A}$.

The simulator then outputs $\mathcal{A}$'s output.

Fig. 4. Simulator for the semi-honest adversary $\mathcal{A}$ corrupting t parties in the set $T \subset \mathcal{P}$.

We will now prove that $\text{IDEAL}_{\mathcal{F}_f,\mathcal{S}_f^{\text{SH}},\mathcal{Z}} \overset{c}{\approx} \text{REAL}_{\Pi_f^{\text{SH}},\mathcal{A},\mathcal{Z}}$ via a series of hybrids. The output of each hybrid is always just the output of the environment $\mathcal{Z}$. Starting with $\mathbf{HYB}_0 = \text{REAL}_{\Pi_f^{\text{SH}},\mathcal{A},\mathcal{Z}}$, we gradually make changes to define $\mathbf{HYB}_1$, $\mathbf{HYB}_2$, $\mathbf{HYB}_3$ and $\mathbf{HYB}_4$.

$\mathbf{HYB}_1$: Same as $\mathbf{HYB}_0$, except that the decryption shares of the honest parties corresponding to the ciphertext $\hat{\mathsf{c}}$ associated with the output wire (obtained after the randomization) are computed using SHE.ShareSim, by inputting to it the decryption shares of the corrupted parties corresponding to $\hat{\mathsf{c}}$, the ciphertext $\hat{\mathsf{c}}$ and the plaintext $\chi(y)$, where y is the function output.

$\mathbf{HYB}_2$: Same as $\mathbf{HYB}_1$, except that $\mathsf{c}_{\underline{\mathbf{1}}}$ obtained from $\mathcal{F}_{\text{SETUPGEN}}$ is replaced by $\mathsf{c}_{\underline{\mathbf{0}}}$ and the circuit is computed as in protocol with $\mathsf{c}_{\underline{\mathbf{0}}}$ being used in place of $\mathsf{c}_{\underline{\mathbf{1}}}$. Moreover, during the randomization step while performing the distributed decryption of the output wire ciphertext, the randomizing ciphertext (c_i, L) of *one* of the *honest* parties (which is an encryption of $\underline{\mathbf{0}}$), say P_h, is replaced by a random encryption of $\chi(y)$.

$\mathbf{HYB}_3$: Same as $\mathbf{HYB}_2$, except that SHE.ShareSim is used while computing the decryption shares of the honest parties for performing the distributed decryption during the evaluation of the refresh gates.

$\mathbf{HYB}_4$: Same as $\mathbf{HYB}_3$, except that the real inputs of the honest parties are replaced by $\chi(0)$ during the **Input Stage** and the circuit is evaluated using encryptions of the $\chi(0)$s as the encrypted inputs of the honest parties.

Our proof will conclude, as we show that every two consecutive hybrids are computationally indistinguishable and $\mathbf{HYB}_4 = \text{IDEAL}_{\mathcal{F}_f,\mathcal{S}_f^{\text{SH}},\mathcal{Z}}$.

$\mathbf{HYB}_0 \overset{c}{\approx} \mathbf{HYB}_1$: This follows from the share simulation indistinguishability property of SHE.

$\mathbf{HYB}_1 \overset{c}{\approx} \mathbf{HYB}_2$: To show the indistinguishability, we rely on the semantic security of SHE. In fact, we use a variant of the semantic security notion, where the adversary gives two pairs of messages to the challenger and the challenger picks a random pair and gives the encryptions for that pair to the adversary. We call this as the *double message* semantic security. It follows by a standard hybrid argument that a scheme offering semantic security also offers double message semantic security with a security loss of a factor of two.

We now show how a distinguisher $\mathcal{Z}$ for the hybrids $\mathbf{HYB}_1$ and $\mathbf{HYB}_2$ can be used to break the double message semantic security of the underlying SHE. Let $\mathcal{R}$ be the attacker that wants to break the double message semantic security of the underlying SHE; $\mathcal{R}$ uses $\mathcal{Z}$ to do so as follows: $\mathcal{R}$ receives the public key pk, evaluation key ek and t keys corresponding to the corrupted parties for performing the distributed decryption. The attacker $\mathcal{R}$ then invokes $\mathcal{Z}$ (in her head), which gives back the input set $(x_1, \ldots, x_n) \in \mathbb{F}_p^n$ for all the parties. Using this output $\mathcal{R}$ computes the function output y and prepares two pairs of messages for the challenger, $(\underline{\mathbf{1}}, \underline{\mathbf{0}})$ and $(\underline{\mathbf{0}}, \chi(y))$ and hands them over to the challenger. Let $\mathcal{R}$ receive back the encrypted pair $(\mathsf{c}', L), (\mathsf{c}, L)$ from the challenger. The algorithm $\mathcal{R}$ now applies SHE.LowerLevel to reduce the first of these to level one, (by abuse of notation we shall still refer to it as c'). Now $\mathcal{R}$ evaluates the circuit by generating offline data honestly and using $(\mathsf{c}', 1)$ in place of $(\mathsf{c}_{\underline{\mathbf{1}}}, 1)$ (that was to be returned by $\mathcal{F}_{\text{SETUPGEN}}$) and (c, L) in place of the randomization ciphertext (namely an encryption of $\underline{\mathbf{0}}$) on the behalf of the honest party P_h (which P_h would have given to randomize the output wire ciphertext). Finally $\mathcal{R}$ outputs what $\mathcal{Z}$ outputs.

It is easy to note that if the challenger had given encryptions of the first pair of messages, namely $(\underline{\mathbf{1}}, \underline{\mathbf{0}})$, then $\mathcal{Z}$ is in $\mathbf{HYB}_1$, else it is in $\mathbf{HYB}_2$. Thus the distinguishing probability of $\mathcal{Z}$ is translated to the winning probability of $\mathcal{R}$ in the double message semantic security game. This implies that our claim is true and there exists no PPT distinguisher $\mathcal{Z}$ for the above two hybrids.

$\mathbf{HYB}_2 \overset{c}{\approx} \mathbf{HYB}_3$: This can be shown by relying on the share simulation indistinguishability property of SHE and by defining $\mathbb{G}_R$ hybrids over the number of refresh gates, where the ith hybrid is same as $\mathbf{HYB}_2$, except that SHE.ShareSim is invoked for the first i refresh gates (assuming topological ordering of the gates) to compute the decryption shares of the honest parties and for the $(i+1)$th refresh gate onwards, the decryption shares of the honest parties are computed as in real protocol using SHE.ShareDec.

$\mathbf{HYB}_3 \overset{c}{\approx} \mathbf{HYB}_4$: We resort to the semantic security of the underlying SHE scheme. We let $H = |\mathcal{P} \setminus T|$ denote the number of honest parties and without loss of generality assume that the first H parties are the honest parties. We introduce $H+1$ hybrids $\mathbf{HYB}_3^0 = \mathbf{HYB}_3, \mathbf{HYB}_3^1, \ldots, \mathbf{HYB}_3^H = \mathbf{HYB}_4$ over the number of honest parties so that the ith hybrid $\mathbf{HYB}_3^i$ is same as the $(i-1)$th hybrid $\mathbf{HYB}_3^{i-1}$, except that the input of the ith honest party is replaced by $\chi(0)$. We now show that $\mathbf{HYB}_3^{i-1} \overset{c}{\approx} \mathbf{HYB}_3^i$ for $i \in [1, \ldots, H]$ which will let us conclude that

$\mathbf{HYB}_3 \overset{c}{\approx} \mathbf{HYB}_4$. We fix an i and show that any $\mathcal{Z}_i$ that tells apart $\mathbf{HYB}_3^{i-1}$ and $\mathbf{HYB}_3^{i}$ can be turned into an attacker that can break semantic security of the SHE scheme.

Let $\mathcal{R}$ be the attacker that wants to break the semantic security of the SHE. The attacker participates in the semantic security game and receives from the challenger pk, ek and t keys corresponding to the corrupted parties for performing the distributed decryption. It then invokes $\mathcal{Z}_i$ (in head) to receive the inputs for the parties, say $(x_1, \ldots, x_n)$ and computes the function output y. The attacker prepares two messages, $\chi(0)$ and $\chi(x_i)$ for the challenger, the latter being received from $\mathcal{Z}_i$ as the input of P_i (namely x_i). In return, the attacker gets back $(\mathfrak{c}_{\mathbf{x}_i}, L)$ which either encrypts $\chi(0)$ or $\chi(x_i)$. Now the attacker computes encryptions of $\chi(0)$ for the first $(i-1)$ parties, for P_i the attacker uses $\mathfrak{c}_{\mathbf{x}_i}$ received from the challenger and for the remaining parties, the attacker computes encryptions of $\chi(x_{i+1}), \ldots, \chi(x_n)$. The attacker $\mathcal{R}$ then honestly evaluates the circuit on these encrypted inputs, ensuring all the similarities between $\mathbf{HYB}_3^{i-1}$ and $\mathbf{HYB}_3^{i}$. Namely, the the attacker performs the offline computation honestly and uses $(\mathfrak{c}_{\underline{\mathbf{0}}}, 1)$ (an encryption of $\underline{\mathbf{0}}$) instead of $(\mathfrak{c}_{\underline{\mathbf{1}}}, 1)$ (as received from the $\mathcal{F}_{\text{SETUPGEN}}$). Moreover, while performing the randomization during the distributed decryption of the output wire ciphertext, the attacker uses an encryption of $\chi(y)$ as the randomizing ciphertext on the behalf of the honest party P_h (instead of an encryption of $\underline{\mathbf{0}}$), so as to make the output wire ciphertext an encryption of $\chi(y)$. Furthermore, the attacker uses $\mathsf{SHE.ShareSim}$ to compute the decryption shares for the honest parties while performing the distributed decryption for the refresh gates and for the output wire. Note that the attacker will know the plaintext associated with the ciphertext to be decrypted (both for the refresh gates as well as for the output wire) while using $\mathsf{SHE.ShareSim}$, even without knowing the actual circuit input of the party P_i (namely the plaintext associated with the challenge ciphertext $(\mathfrak{c}_{\mathbf{x}_i}, L)$) used for the circuit evaluation. This is because now $\mathfrak{c}_{\underline{\mathbf{0}}}$ (instead of $c_{\underline{\mathbf{1}}}$) is multiplied with the ciphertexts that are to be decrypted in the protocol and so the post-multiplication ciphertexts have associated plaintext $\underline{\mathbf{0}}$, irrespective of the actual circuit inputs. This allows $\mathcal{R}$ to invoke $\mathsf{SHE.ShareSim}$ on a ciphertext for which it knows the associated plaintext even without knowing the inputs to the circuit. More specifically, for every refresh gate, $\mathcal{R}$ now knows the plaintext associated with the ciphertext to be decrypted, since it solely depends on the data created in offline computation which will be known to $\mathcal{R}$. On the other hand, for the output wire, $\mathcal{R}$ knows the plaintext associated with the ciphertext to be decrypted, since it is nothing but the circuit output $\chi(y)$. Finally at the end of the circuit evaluation as above, $\mathcal{R}$ outputs what $\mathcal{Z}_i$ outputs.

Now note that if the challenge ciphertext $(\mathfrak{c}_{\mathbf{x}_i}, L)$ is an encryption of $\chi(x_i)$, then $\mathcal{Z}_i$ is in $\mathbf{HYB}_3^{i-1}$, else it is in $\mathbf{HYB}_3^{i}$. The above reduction thus shows that $\mathcal{R}$ can distinguish between encryptions of $\chi(x_i)$ and $\chi(0)$ with the same probability with which $\mathcal{Z}_i$ can distinguish between $\mathbf{HYB}_3^{i-1}$ and $\mathbf{HYB}_3^{i}$. This implies that our claim is true.

$\mathbf{HYB}_4 \overset{s}{\approx} \mathbf{IDEAL}_{\mathcal{F}_f, \mathcal{S}_f^{\text{SH}}, \mathcal{Z}}$: Follows from the inspection that the following steps have been performed in $\mathbf{HYB}_4$ as well $\text{IDEAL}_{\mathcal{F}_f, \mathcal{S}_f^{\text{SH}}, \mathcal{Z}}$: (1) $\mathfrak{c}_{\underline{\mathbf{1}}}$ is replaced by $\mathfrak{c}_{\underline{\mathbf{0}}}$, (2) the inputs of the honest parties are replaced by $\chi(0)$s, (3) $\mathsf{SHE.ShareSim}$ is invoked to compute the decryption shares of the honest parties corresponding to all the refresh gates as well as in the output computation stage and (4) One of the honest party's randomizing ciphertext is an encryption of $\chi(y)$ instead of an encryption of $\underline{\mathbf{0}}$.

Thus we have proved the following claim that in turn concludes the theorem.

Claim. $\text{IDEAL}_{\mathcal{F}_f, \mathcal{S}_f^{\text{SH}}, \mathcal{Z}} \overset{c}{\approx} \text{REAL}_{\Pi_f^{\text{SH}}, \mathcal{A}, \mathcal{Z}}$.

□

5 MPC from SHE – The Active Setting

The functionalities from Section 4 are in the passive corruption model. In the presence of an active adversary, the functionalities will be modified as follows: the respective functionality considers the input received from the majority of the parties and performs the task it is supposed to do on those inputs. For example, in the case of $\mathcal{F}_f$, the functionality considers for the computation those x_is, corresponding to the P_is from which the functionality has received the message (sid, i, x_i); on the behalf of the remaining P_is, the functionality substitutes 0 as the default input for the computation. Similarly for $\mathcal{F}_{\text{SETUPGEN}}$, the functionality performs its task if it receives the message (sid, i) from the majority of the parties. These are the standard notions of defining ideal functionalities for various corruption scenarios and we refer [32] for the complete formal details; we will not present separately the ideal functionality $\mathcal{F}_f$ and $\mathcal{F}_{\text{SETUPGEN}}$ for the malicious setting.

A closer look at Π_f^{SH} shows that we can "compile" it into an actively secure MPC protocol tolerating t active corruptions if we ensure that every corrupted party "proves" in a zero knowledge (ZK) fashion that it constructed the following correctly: **(1)** The ciphertexts in the offline phase; **(2)** The ciphertexts during the input stage and **(3)** The randomizing ciphertexts during the output stage.

Apart from the above three requirements, we also require a "robust" version of the $\mathsf{SHE.ShareCombine}$ method which works correctly even if up to t input decryption shares are incorrect. In Section 6 we show that for our specific SHE scheme, the $\mathsf{SHE.ShareCombine}$ algorithm (based on the standard error-correction) is indeed robust, provided $t < n/3$. For the case of $t < n/2$ we also show that by including additional steps and zero-knowledge proofs (namely proof of correct decryption), one can also obtain a robust output. Interestingly the MPC protocol requires the transmission of at most $\mathcal{O}(n^3)$ such additional zero-knowledge proofs; i.e. the communication needed to obtain robustness is *independent* of the circuit. We stress that $t < n/2$ is the *optimal resilience* for computationally secure MPC against active corruptions (with robustness and fairness) [15, 33]. To keep the protocol presentation and its proof simple, we assume a robust $\mathsf{SHE.ShareCombine}$ (i.e. for the case of $t < n/3$), which applies error correction for the correct decryption. In the same section, we further present a more efficient offline phase attaining a linear communication overhead (asymptotically) in the number of preprocessed ciphertexts.

Functionality $\mathcal{F}_{\mathbf{ZK}}^R$

$\mathcal{F}_{\text{ZK}}^R$ interacts with a prover $P_i \in \{P_1, \ldots, P_n\}$ and the set of n verifiers $\mathcal{P} = \{P_1, \ldots, P_n\}$ and the adversary $\mathcal{S}$.

- Upon receiving $(\mathsf{sid}, i, (x, w))$ from the prover $P_i \in \{P_1, \ldots, P_n\}$, the functionality sends (sid, i, x) to all the verifiers in $\mathcal{P}$ and $\mathcal{S}$ if $R(x, w)$ is true. Else it sends $(\mathsf{sid}, i, \perp)$ and halts.

Fig. 5. The Ideal Functionality for ZK

The actively secure MPC protocol is given in Figure 5, it uses an ideal ZK functionality $\mathcal{F}_{\text{ZK}}^R$, parametrized with an NP-relation R. We apply this ZK functionality to the following relations to obtain the functionalities $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ and $\mathcal{F}_{\text{ZK}}^{R_{zeroenc}}$. We note that UC-secure realizations of $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ and $\mathcal{F}_{\text{ZK}}^{R_{zeroenc}}$ can be obtained in the CRS model, similar techniques to these are used in [2]. Finally we do not worry about the instantiation of $\mathcal{F}_{\text{SETUPGEN}}$ as we consider it a one time set-up, which can be done via standard techniques (such as running an MPC protocol).

- $R_{enc} = \{((\mathfrak{c}, \mathfrak{l}), (\mathbf{x}, r)) \quad | \quad (\mathfrak{c}, \mathfrak{l}) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{x}, r) \text{ if } \mathfrak{l} = L \quad \vee \quad (\mathfrak{c}, \mathfrak{l}) = \mathsf{SHE.LowerLevel}_{\mathsf{ek}}(\mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{x}, r), 1) \text{ if } \mathfrak{l} = 1\}$: we require this relation to hold for the offline stage ciphertexts (where $\mathfrak{l} = L$) and for the input stage ciphertexts (where $\mathfrak{l} = 1$).
- $R_{zeroenc} = \{((\mathfrak{c}, L), (\mathbf{x}, r)) \quad | \quad (\mathfrak{c}, L) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{x}, r) \wedge \mathbf{x} = \underline{\mathbf{0}}\}$: we require this relation to hold for the randomizing ciphertexts during the output stage.

We are now ready to present the protocol Π_f^{MAL} (see Figure 6) in the $(\mathcal{F}_{\text{SETUPGEN}}, \mathcal{F}_{\text{ZK}}^{R_{enc}}, \mathcal{F}_{\text{ZK}}^{R_{zeroenc}})$-hybrid model and assuming a robust $\mathsf{SHE.ShareCombine}$ based on error-correction (i.e. for the case $t < n/3$).

Theorem 2. *Let $f : \mathbb{F}_p^n \to \mathbb{F}_p$ be a function represented by a well-formed arithmetic circuit C over $\mathbb{F}_p$. Let $\mathcal{F}_f$ (presented in Figure 1) be the ideal functionality computing f and let* SHE *be a threshold L-levelled SHE scheme such that* $\mathsf{SHE.ShareCombine}$ *is robust. Then the protocol Π_f^{MAL} UC-secure realises $\mathcal{F}_f$ in the $(\mathcal{F}_{\text{SETUPGEN}}, \mathcal{F}_{\text{ZK}}^{R_{enc}}, \mathcal{F}_{\text{ZK}}^{R_{zeroenc}})$-hybrid Model against a static, active adversary $\mathcal{A}$ corrupting t parties.*

Proof. Since the robust $\mathsf{SHE.ShareCombine}$ works correctly even in the presence of t active corruptions, the correctness of our MPC protocol follows from the properties of $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ and $\mathcal{F}_{\text{ZK}}^{R_{zeroenc}}$ by using the same arguments as used in Theorem 1. More specifically, the properties of $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ ensures that during the offline computation, each *corrupted* P_i knows the plaintext $\mathbf{m}_{ik}$ associated with the ciphertext $\mathfrak{c}_{\mathbf{m}_{ik}}$. Due to the same reason, each corrupted P_i knows the plaintext (namely the input) $\chi(x_i)$ associated with the ciphertext $\mathfrak{c}_{\mathbf{x}_i}$. Moreover, the property of $\mathcal{F}_{\text{ZK}}^{R_{zeroenc}}$ ensures that

Protocol Π_f^{MAL}

Let C be the well formed arithmetic circuit over $\mathbb{F}_p$ representing the function f, let C^{aug} denote an augmented circuit associated with C, and let SHE be a threshold L-levelled SHE scheme. For session ID sid the parties in $\mathcal{P} = \{P_1, \ldots, P_n\}$ do the following:

Offline Computation: Every party $P_i \in \mathcal{P}$ does the following:

- Call $\mathcal{F}_{\text{SETUPGEN}}$ with (sid, i) and receive $(\mathsf{sid}, \mathsf{pk}, \mathsf{ek}, \mathsf{dk}_i, (\mathfrak{c}_{\underline{\mathbf{1}}}, 1))$.
- Same as in the offline phase of Π_f^{SH}, except that for every message $\mathbf{m}_{ik}$ for $k \in [1, \ldots, \zeta]$ and the corresponding ciphertext $(\mathfrak{c}_{\mathbf{m}_{ik}}, L) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{m}_{ik}, r_{ik})$, call $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ with $(\mathsf{sid}, i, ((\mathfrak{c}_{\mathbf{m}_{ik}}, L), (\mathbf{m}_{ik}, r_{ik})))$. Receive $(\mathsf{sid}, j, (\mathfrak{c}_{\mathbf{m}_{jk}}, L))$ from $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ for $k \in [1, \ldots, \zeta]$ corresponding to each $P_j \in \mathcal{P}$. If $(\mathsf{sid}, j, \perp)$ is received from $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ for some $P_j \in \mathcal{P}$, then consider ζ publicly known level L encryptions of random values from $\mathcal{M}$ as $(\mathfrak{c}_{\mathbf{m}_{jk}}, L)$ for $k \in [1, \ldots, \zeta]$.

Online Computation: Every party $P_i \in \mathcal{P}$ does the following:

- **Input Stage**: On having input $x_i \in \mathbb{F}_p$, compute level L ciphertext $(\mathfrak{c}_{\mathbf{x}_i}, 1) = \mathsf{SHE.LowerLevel}_{\mathsf{ek}}(\mathsf{SHE.Enc}_{\mathsf{pk}}(\chi(x_i), r_i), 1)$ with randomness r_i and call $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ with the message $(\mathsf{sid}, i, ((\mathfrak{c}_{\mathbf{x}_i}, 1), (\chi(x_i), r_i)))$. Receive $(\mathsf{sid}, j, (\mathfrak{c}_{\mathbf{x}_j}, 1))$ from $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ corresponding to each $P_j \in \mathcal{P}$. If $(\mathsf{sid}, j, \perp)$ is received from $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ for some $P_j \in \mathcal{P}$, then consider a publicly known level 1 encryption of $\chi(0)$ as $(\mathfrak{c}_{\mathbf{x}_j}, 1)$ for such a P_j.
- **Computation Stage**: Same as Π_f^{SH}, except that now the robust $\mathsf{SHE.ShareCombine}$ is used.
- **Output Stage**: Let $(\mathfrak{c}, \mathfrak{l})$ be the ciphertext associated with the output wire of C^{aug} where $\mathfrak{l} \in [1, \ldots, L]$.
  - **Randomization:** Compute a random encryption $(c_i, L) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\underline{\mathbf{0}}, r_i')$ of $\underline{\mathbf{0}} = (0, \ldots, 0)$ and call $\mathcal{F}_{\text{ZK}}^{R_{zeroenc}}$ with the message $(\mathsf{sid}, i, ((c_i, L), (\underline{\mathbf{0}}, r_i')))$. Receive $(\mathsf{sid}, j, (c_j, L))$ from $\mathcal{F}_{\text{ZK}}^{R_{zeroenc}}$ corresponding to each $P_j \in \mathcal{P}$. If $(\mathsf{sid}, j, \perp)$ is received from $\mathcal{F}_{\text{ZK}}^{R_{zeroenc}}$ for some $P_j \in \mathcal{P}$, then consider a publicly known level L encryption of $\underline{\mathbf{0}}$ as (c_j, L) for such a P_j.
  - The rest of the steps are same as in Π_f^{SH}, except that now the robust $\mathsf{SHE.ShareCombine}$ is used.

Fig. 6. The Protocol for Realizing $\mathcal{F}_f$ against an Active Adversary in the $(\mathcal{F}_{\text{SETUPGEN}}, \mathcal{F}_{\text{ZK}}^{R_{enc}}, \mathcal{F}_{\text{ZK}}^{R_{zeroenc}})$-hybrid Model

each corrupted P_i has indeed contributed an encryption of $\underline{\mathbf{0}}$ as a randomizing ciphertext during the distributed decryption of the output wire ciphertext. The homomorphic property of the SHE ensures that the addition and multiplication gates are evaluated correctly. We next argue that even the refresh gates are evaluated correctly. This follows because once the parties have access to the offline data, each refresh gate can be evaluated correctly if the parties are able to decrypt the corresponding masked ciphertext $\mathfrak{c}_{\mathbf{z}+\mathbf{m}}$. However since $\mathsf{SHE.ShareCombine}$ works even in the presence of t active corruptions, it follows that the parties can decrypt $\mathfrak{c}_{\mathbf{z}+\mathbf{m}}$. Due to the same reason, the parties will be able to decrypt the ciphertext associated with the output wire and hence can obtain the function output.

We next prove the security. Let $\mathcal{A}$ be a real-world active adversary up to t parties and let $T \subset \mathcal{P}$ denote the set of corrupted parties. We now present an ideal-world adversary (simulator) $\mathcal{S}_f^{\text{MAL}}$ for $\mathcal{A}$ in Figure 7; for simplicity, we assume that an SHE with a robust, non-interactive $\mathsf{SHE.ShareCombine}$ (i.e. for $t < n/3$) has been used in the MPC protocol. The indistinguishability between the real and ideal world now follows mostly by the similar arguments given for semi-honest case (see the proof of Theorem 1).

□

Simulator $\mathcal{S}_f^{\text{MAL}}$

Let SHE be a threshold L-levelled SHE scheme. The simulator plays the role of the honest parties and simulates each step of the protocol Π_f^{MAL} as follows. The communication of the $\mathcal{Z}$ with the adversary $\mathcal{A}$ is handled as follows: Every input value received by the simulator from $\mathcal{Z}$ is written on $\mathcal{A}$'s input tape. Likewise, every output value written by $\mathcal{A}$ on its output tape is copied to the simulator's output tape (to be read by the environment $\mathcal{Z}$). The simulator then does the following for session ID sid:

Offline Computation:

- On receiving the message (sid, i) to $\mathcal{F}_{\text{SETUPGEN}}$ from $\mathcal{A}$ for each $P_i \in T$, invoke $(\mathsf{pk}, \mathsf{ek}, \mathsf{sk}, \mathsf{dk}_1, \ldots, \mathsf{dk}_n) = \mathsf{SHE.KeyGen}(1^\kappa, n)$, compute $(\mathfrak{c}_{\underline{\mathbf{0}}}, 1) = \mathsf{SHE.LowerLevel}_{\mathsf{ek}}(\mathsf{SHE.Enc}_{\mathsf{pk}}(\underline{\mathbf{0}}, \cdot), 1)$, and send $(\mathsf{sid}, \mathsf{pk}, \mathsf{ek}, \{\mathsf{dk}_i\}_{P_i \in T}, (\mathfrak{c}_{\underline{\mathbf{0}}}, 1))$ to $\mathcal{A}$.
- For each party $P_j \notin T$ and $k \in [1, \ldots, \zeta]$, compute $(\mathfrak{c}_{\mathbf{m}_{jk}}, L) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{m}_{jk}, \cdot)$ for a randomly chosen $\mathbf{m}_{jk} \in \mathcal{M}$ and send $(\mathsf{sid}, j, (\mathfrak{c}_{\mathbf{m}_{jk}}, L))$ to $\mathcal{A}$ on the behalf of $\mathcal{F}_{\text{ZK}}^{R_{enc}}$. For each $P_i \in T$ on receiving $(\mathsf{sid}, i, (\mathfrak{c}_{\mathbf{m}_{ik}}, L), (\mathbf{m}_{ik}, r_{ik}))$ as a message to $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ from $\mathcal{A}$ for $k \in [1, \ldots, \zeta]$, verify if $(\mathfrak{c}_{\mathbf{m}_{ik}}, L) \stackrel{?}{=} \mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{m}_{ik}, r_{ik})$. If the verification fails for some $P_i \in T$ then send $(\mathsf{sid}, i, \perp)$ ζ times (corresponding to ζ ciphertexts) to $\mathcal{A}$ and set ζ publicly known level L encryptions of random values from $\mathcal{M}$ as $(\mathfrak{c}_{\mathbf{m}_{ik}}, L)$ for $k \in [1, \ldots, \zeta]$. Compute the kth ciphertext and the kth plaintext of the offline phase as in Π_f^{MAL}. The later can be computed by the simulator since it knows all the plaintexts.

Online Computation:

- **Input Stage**:
  - For every party $P_j \in \mathcal{P} \setminus T$, compute a random encryption $(\mathfrak{c}_{\mathbf{x}_j}, 1) = \mathsf{SHE.LowerLevel}_{\mathsf{ek}}(\mathsf{SHE.Enc}_{\mathsf{pk}}(\chi(0), \cdot), 1)$ and send $(\mathsf{sid}, j, (\mathfrak{c}_{\mathbf{x}_j}, 1))$ to $\mathcal{A}$ on the behalf of $\mathcal{F}_{\text{ZK}}^{R_{enc}}$. For each $P_i \in T$ on receiving $(\mathsf{sid}, i, (\mathfrak{c}_{\mathbf{x}_i}, 1), (\chi(x_i), r_i))$ as a message to $\mathcal{F}_{\text{ZK}}^{R_{enc}}$ from $\mathcal{A}$, verify $(\mathfrak{c}_{\mathbf{x}_i}, 1) \stackrel{?}{=} \mathsf{SHE.LowerLevel}_{\mathsf{ek}}(\mathsf{SHE.Enc}_{\mathsf{pk}}(\chi(x_i), r_i))$ and send $(\mathsf{sid}, i, \perp)$ to $\mathcal{A}$ if verification fails. Use publicly known ciphertext $(\mathfrak{c}_{\mathbf{x}_i}, 1)$ encrypting $x_i = \chi(0)$ on the behalf of any such P_i.
  - Send (sid, i, x_i) to $\mathcal{F}_f$ on the behalf of each $P_i \in T$ and receive the function output y.
- **Computation Stage**: The simulator acts in the same way as in $\mathcal{S}_f^{\text{SH}}$ except that whenever $\mathcal{A}$ sends the decryption shares corresponding to the parties in T during the evaluation of the refresh gates, the simulator ignores them; instead it computes the decryption shares by itself using the keys dk_i (for the distributed decryption) corresponding to $P_i \in T$ (the simulator knows dk_i for every $P_i \in T$ since it generated them by itself). These new decryption shares are then fed to $\mathsf{SHE.ShareSim}$ to obtain the simulated decryption shares corresponding to the honest parties, which the simulator then sends to $\mathcal{A}$ on behalf of the honest parties.
- **Output Stage**:
  - **Randomization**: Let $H = \mathcal{P} \setminus T$ be the set of honest parties and let P_h be some party in H. For every $P_j \in H \setminus \{P_h\}$ compute a random encryption $(\mathfrak{c}_j, L) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\underline{\mathbf{0}}, \cdot)$, while for $P_h \in H$ compute a random encryption $(\mathfrak{c}_h, L) = \mathsf{SHE.Enc}_{\mathsf{pk}}(\chi(y), \cdot)$. For every $P_j \in H$, send $(\mathsf{sid}, j, (\mathfrak{c}_j, L))$ to $\mathcal{A}$ on the behalf of $\mathcal{F}_{\text{ZK}}^{R_{zeroenc}}$.
  - For each $P_i \in T$ on receiving $(\mathsf{sid}, i, (\mathfrak{c}_i, L), (\underline{\mathbf{0}}, r_i'))$ as a message to $\mathcal{F}_{\text{ZK}}^{R_{zeroenc}}$ from $\mathcal{A}$, verify if $(\mathfrak{c}_i, L) \stackrel{?}{=} \mathsf{SHE.Enc}_{\mathsf{pk}}(\underline{\mathbf{0}}, r_i')$. If the verification fails for some $P_i \in T$ then send $(\mathsf{sid}, i, \perp)$ to $\mathcal{A}$ and consider a publicly known level L encryption of $\underline{\mathbf{0}}$ as $(\mathfrak{c}_i, L)$ for such a P_i.
  - On receiving the decryption shares from $\mathcal{A}$ corresponding to the parties $P_i \in T$, the simulator ignores them and instead recomputes them using the dk_is and feed them to $\mathsf{SHE.ShareSim}$ to compute the simulated decryption shares for the honest parties. Finally it sends the simulated shares to $\mathcal{A}$ on behalf of the honest parties.

The simulator then outputs $\mathcal{A}$'s output.

Fig. 7. Simulator for the active adversary $\mathcal{A}$ corrupting t parties in the set $T \subset \mathcal{P}$.

6 Obtaining a Robust Protocol

In this section we discuss how to achieve a robust $\mathsf{SHE.ShareCombine}$ for our precise SHE scheme, then we present a modified offline phase with linear communication overhead.

Recall that in our concrete SHE scheme, the $\mathsf{SHE.ShareCombine}$ algorithm takes as input a set of shares obtained via Shamir Secret sharing over the ring R_{q_0}. From this observation it is clear, by the standard error correction properties of the Reed-Solomon codes (upon which the Shamir secret sharing is based), that one can obtain a robust $\mathsf{SHE.ShareCombine}$ algorithm immediately in the case of $t < n/3$.

All that remains is to present a robust $\mathsf{SHE.ShareCombine}$ for the case $t < n/2$. We present the protocol (note that $\mathsf{SHE.ShareCombine}$ will be now a protocol instead of a local algorithm as it may involve interaction among the parties) in Figure 8 that uses the dispute-control framework proposed in [3] and the fact that Reed-Solomon codes can detect up to $t < n/2$ errors. The protocol also invokes the ZK functionality for the relation $R_{sharedec}$ a limited number of times for the proof of correct (distributed) decryption, where $R_{sharedec}$ is given below.

$$R_{sharedec} = \{(((\mathfrak{c}, \mathfrak{l}), \bar{\mu}_i), \mathsf{dk}_i) \quad | \quad \bar{\mu}_i = \mathsf{SHE.ShareDec}_{\mathsf{dk}_i}(\mathfrak{c}, \mathfrak{l})\}$$

Unlike the functionality $\mathcal{F}_{\text{ZK}}^{R}$ defined in Figure 5 that treats all the parties in $\mathcal{P}$ as the verifiers, it is enough if the functionality for $R_{sharedec}$ is defined in a single prover and a single verifier setting. However we avoid elaborating more on this to keep simplicity.

Our robust $\mathsf{SHE.ShareCombine}$ realises the following idea: For distributed decryption, as usual, every party sends the decryption shares to every other party. A party P_i on receiving the decryption shares first check whether all of them lie on a unique polynomial of degree at most t (namely error detection). If no error is detected then the secret can be safely reconstructed. However if some error is detected then P_i "complains" to the parties, asking them to prove the correctness of their respective decryption shares sent earlier; the parties respond back with ZK proofs by calling the $\mathcal{F}_{\text{ZK}}^{R_{sharedec}}$ functionality. Now P_i can "identify" the incorrect decryption share providers and ignore their shares in the future instances of distributed decryption. Each party P_i keeps a list $\mathcal{H}_i$ of the parties who it believes to be honest so far. Proper care has to be taken to ensure that the honest parties do not respond back "too many times" to the "false" complaints issued by the corrupted parties. This is resolved via keeping counters for the complaints. The idea is that an honest P_j will complain to an honest P_i at most t times and thus all the complaints from P_j after tth complaint clearly indicates that the complaint is false and P_j is corrupted. It is now easy to see that by using this trick, the *total* number of calls to $\mathcal{F}_{\text{ZK}}^{R_{sharedec}}$ in the MPC protocol will be $\mathcal{O}(n^3)$, which is independent of the circuit size; this is because a party may have to provide ZK proof to another party (by calling $\mathcal{F}_{\text{ZK}}^{R_{sharedec}}$) in at most t instances of distributed decryption. For large circuit sizes the extra communication cost to obtain a robust $\mathsf{SHE.ShareCombine}$ in the case $n/3 \leq t < n/2$ can be safely ignored.

SHE.ShareCombine

Each party P_i maintains its local copy $\mathcal{H}_i$ of a list all the parties which it currently assumed to be honest. Initially each $\mathcal{H}_i = \{P_1, \ldots, P_n\}$. Apart from this, every party P_i maintains n counters $\mathsf{cnt}_{i,1}, \ldots, \mathsf{cnt}_{i,n}$, where $\mathsf{cnt}_{i,j}$ is used to maintain a count of number of times an error message has been received from the party P_j; initially all these counters are set to 0. To execute an $\mathsf{SHE.ShareCombine}((\mathfrak{c}, \mathfrak{l}), \{\bar{\mu}_j\}_{j\in\{1\ldots,n\}})$ operation, where $\bar{\mu}_j$ has been sent by P_j, party P_i performs the following steps:

- Ignore all $\bar{\mu}_j$ where $P_j \notin \mathcal{H}_i$. If the remaining $\bar{\mu}_j$s lie on a unique polynomial of degree at most t, then output the corresponding secret (namely the constant term of the polynomial). Otherwise, send a message $(\mathsf{sid}, i, \mathtt{Error}_i, (\mathfrak{c}, \mathfrak{l}))$ to every party $P_j \in \mathcal{H}_i$.
- If an error message $(\mathsf{sid}, j, \mathtt{Error}_j, (\mathfrak{c}, \mathfrak{l}))$ has been received from some $P_j \in \mathcal{H}_i$ then check whether $\mathsf{cnt}_{i,j} < t$. If $\mathsf{cnt}_{i,j} < t$, then call $\mathcal{F}_{\mathrm{ZK}}^{R_{sharedec}}$ with the message $(\mathsf{sid}, i, j, (((\mathfrak{c}, \mathfrak{l}), \bar{\mu}_i), \mathsf{dk}_i))$ and set $\mathsf{cnt}_{i,j} := \mathsf{cnt}_{i,j} + 1$. Else if $\mathsf{cnt}_{i,j} \geq t$ then remove P_j from the list $\mathcal{H}_i$.
- If an error message $(\mathsf{sid}, i, \mathtt{Error}_i, (\mathfrak{c}, \mathfrak{l}))$ has been sent in the first step, then execute the following: receive $(\mathsf{sid}, j, i, ((\mathfrak{c}, \mathfrak{l}), \bar{\mu}_j))$ from $\mathcal{F}_{\mathrm{ZK}}^{R_{sharedec}}$ for every $P_j \in \mathcal{H}_i$. If for some $P_j \in \mathcal{H}_i$, the message $(\mathsf{sid}, j, i, \perp)$ is received from $\mathcal{F}_{\mathrm{ZK}}^{R_{sharedec}}$ then remove P_j from $\mathcal{H}_i$. Using the $\bar{\mu}_j$s corresponding to the $P_j \in \mathcal{H}_i$, interpolate the polynomial of degree at most t, output its constant term as the secret.

Fig. 8. Robust $\mathsf{SHE.ShareCombine}$ For $t < n/2$

6.1 An Improved Offline Phase (sketch)

From the analysis in Section 9, we find that the online communication complexity of our protocol is $\mathsf{Cost} = \mathcal{O}(n \cdot |\mathbb{G}_M|)$ (in the asymptotic sense). We now sketch that how we can modify our offline computation so that asymptotically the communication complexity of the offline phase is $\mathcal{O}(n \cdot \zeta)$, where $\zeta > \mathbb{G}_R$ is the number of random ciphertexts generate in the offline phase. We need the following three tools:

- Multi-valued Broadcast with $\mathcal{O}(n)$ Overhead [27]: This protocol allows a sender $\mathsf{Sen} \in \{P_1, \ldots, P_n\}$ to send a message m of size ℓ "identically" to all the n parties (even if Sen is corrupted). The protocol can tolerate up to $t < n/2$ faults (even if the adversary is computationally unbounded) and has communication complexity $\mathcal{O}(n\ell)$ provided $\ell = \Omega(n^3)$.
- Randomness Extraction [33, 24]: Given a set of n encryptions of random values t of which may be known to the adversary, the randomness extraction algorithm based on superinvertible matrix [33] or Vandermonde matrix [24] allows the parties to (locally) compute encryptions of $(n - t)$ random values unknown to the adversary.
- Non-interactive Zero Knowledge Proofs: We require UC-secure instantiation of $\mathcal{F}_{\mathrm{ZK}}^{R_{enc}}$, such that a party $P_i \in \{P_1, \ldots, P_n\}$ on computing encryptions of ℓ random values can publicly prove to anyone that it knows the associated plaintexts by "attaching" a proof of size $\mathcal{O}(\ell)$. Such proofs can be obtained, for example using the techniques of [2].

Now the offline phase protocol will proceed as follows: every party P_i computes encryptions of $\mathcal{L}$ random elements along with a NIZK proof that it knows the associated plaintexts where $\mathcal{L} = \frac{\zeta}{(n-t)}$. Party P_i then broadcasts the ciphertexts along with the proof by acting as a Sen and invoking the instance of a multi-valued broadcast protocol. The ciphertexts received from the different parties are then perceived as $\mathcal{L}$ batches of ciphertexts, where the lth batch consists of the lth ciphertext broadcasted by each party for $l \in [1, \ldots, \mathcal{L}]$. Finally, the randomness extraction algorithm on each batch of ciphertext to obtain $(n-t)$ random ciphertexts from each batch and in total $\mathcal{L} \cdot (n-t) = \zeta$ ciphertexts. Assuming $\mathcal{L} = \Omega(n^3)$, the total communication cost for the offline phase is now $\mathcal{O}(n \cdot \zeta)$: each instance of broadcast protocol has communication complexity $\mathcal{O}(n \cdot \mathcal{L}) = \mathcal{O}(\mathcal{L})$, as $(n - t) = \Theta(n)$. It is easy to see that the output ciphertexts are indeed random as there exists at least $(n - t)$ honest parties corresponding to each batch of ciphertexts. Note that we do not require any powerful (but somewhat complex) tools like player elimination, as used in the MPC protocol of [33] (whose communication complexity is also $\mathcal{O}(n \cdot \zeta)$).

7 Instantiating our FHE using BGV

In this section we show an instantiation of SHE based on the scheme of Brakerski, Gentry and Vaikuntanathan (BGV) ([10]). As in [6] we make use of Shamir secret sharing to share the secret key among the parties and pseudorandom secret sharing (PRSS) [17] to non-interactively share a pseudorandom value from a *chosen* interval. We describe a variant of the BGV-type cryptosystems based on the ring learning with error (RLWE) assumption ([36]), naturally supporting the packing operations described in Section 3.

7.1 Preliminaries

Plaintext Space: We define the polynomial ring $R := \mathbb{Z}[x]/(f(x))$, where $f(x)$ is a monic irreducible polynomial. For our purposes it will suffice to fix $f(x)$ as the cyclotomic polynomial $\Phi_m(x) = x^{m/2} + 1$ with m a power of two. We set $N = \phi(m) = m/2$, where ϕ is the Euler totient function. The ring R is the ring of integers of the mth cyclotomic number field $\mathbb{Q}(\zeta_m)$, with ζ_m an mth root of unity. Denote by $R_q := R/qR$, for an integer q the reduction of R modulo q, i.e. the set of all integer polynomials of degree at most $N-1$ with coefficients in $(-q/2, q/2]$.

Looking ahead the plaintext space of the scheme will be defined to be $R_p := R/pR$ for some prime p such that $p \equiv 1 \mod m$. Since $p \equiv 1 \pmod m$, the polynomial $\Phi_m(x)$ splits into distinct linear factors $F_i(x)$ modulo p:

$$\mathcal{M} := R_p \cong \mathbb{Z}_p[x]/F_1(x) \times \cdots \times \mathbb{Z}_p[x]/F_N(x) \cong \mathbb{F}_p^N,$$

where each factor corresponds to an independent "plaintext slot", holding an element of the finite field $\mathbb{F}_p$. Thus each message $\mathbf{m} \in \mathcal{M}$ actually corresponds to N messages in $\mathbb{F}_p$ and can be represented as an N-vector $(\mathbf{m} \mod F_i)_{i=1,\ldots,N}$. By the Chinese Remainder Theorem addition and multiplication in R_p correspond to SIMD (Single Instruction Multiple Data) operations on the slots and this allows to process N input values at once as described in Section 3.

If we consider the Galois group Gal of $\mathbb{Q}(\zeta_m)$, then $\mathsf{Gal} = \mathsf{Gal}(\mathbb{Q}(\zeta_m)/\mathbb{Q}) \cong \mathbb{Z}_m^*$ and it is formed by the mappings $\sigma_i : a(x) \mapsto a(x^i) \mod \Phi_m(x)$ for all $i \in \mathbb{Z}_m^*$. It is well known ([30]) that Gal transitively acts on plaintext slots, i.e. $\forall i, j \in \{1, \ldots, N\}$ there exists an element $\sigma_{i \to j} \in \mathsf{Gal}$ which sends an element in slot i to an element in slot j.

Random Values: During our construction we will need to sample elements from different distributions over R_q. We will use the following distributions over R, and then map to R_q as appropriate.

- $\mathcal{HWT}(h, N)$: This generates a vector of length N with elements chosen from $\{-1, 0, 1\}$ such that the number of non-zero elements is equal to h.
- $\mathcal{ZO}(0.5, N)$: This generates a vector of length N with elements chosen from $\{-1, 0, 1\}$ such that the coefficient probabilities are $p_{-1} = 1/4$, $p_0 = 1/2$ and $p_1 = 1/4$.
- $\mathcal{DG}(\sigma^2, N)$: This generates a vector of length N with elements chosen according to the discrete Gaussian distribution $D_{\mathbb{Z}^N, \sigma}$.
- $\mathcal{RC}(0.5, \sigma^2, N)$: This generates a triple of elements (a, b, c) where a is sampled from $\mathcal{ZO}_s(0.5, N)$ and b and c are sampled from $\mathcal{DG}_s(\sigma^2, N)$.
- $\mathcal{U}(q, N)$: This generates a vector of length N with elements generated uniformly modulo q.

Pseudorandom Secret Sharing Over Polynomial Rings: Pseudorandom secret sharing was first introduced in [17]. Given a setup, a PRSS scheme allows parties to generate almost unlimited number of Shamir sharings of pseudorandom values at the cost of no communication. Furthermore, the setup is generated once and for all and therefore can be reused many times. While known PRSS works over fields or rings [17, 6], for our purposes we will require a PRSS defined over the polynomial rings R_{q_l}.

In [17] the construction of a PRSS was presented. This was used in [6] to construct a PRSS over $\mathbb{Z}_q$, where $q = \prod p_i$ for n parties, such that each p_i is prime with $p_i > n$. This construction immediately extends to R_q by computing the underlying PRF N times. For completeness we overview the construction here: Given an element $s \in R_q$, we use $[s]$ for the Shamir's sharing of s, $[s]_i = s_i$ for the ith component of the sharing of s, $i = 1, \ldots, n$. We

assume a prior one-time setup which distributes a vector of shared keys $\mathbf{k}_A = (\mathsf{k}_{0,A}, \ldots, \mathsf{k}_{N-1,A})$ to each party in A for every subset A of size $n-t$. These keys will be used as the keys of a keyed pseudorandom function PRF family, $\{\psi_{\mathsf{k}}(\cdot)\}_{\mathsf{k}\in\mathbb{K}}$. The pseudorandomness of the output of the following algorithm can be reduced to the PRF security of the underlying PRF at the cost of security loss by a factor of $1/N$.

1. The parties in $\mathcal{P}$ agree on N elements $t_j \in \mathbb{Z}_q$ for $j \in \{0, \ldots, N-1\}$.
2. For $j = 0, \ldots, N-1$, every party $P_i \in \mathcal{P}$ computes $[s_j]_i = \sum_{A\subset\mathcal{P}:|A|=n-t, P_i\in A} \psi_{\mathsf{k}_{j,A}}(t_j)\cdot f_A(i)$. Where $f_A(X)$ denotes the polynomial of degree at most t, such that $f_A(0) = 1$ and $f_A(l) = 0$ for every $P_l \notin A$.
3. For $j = 0, \ldots, N-1$, the value $s_j = \sum_{A\subset\mathcal{P}:|A|=n-t, P_i\in A} \psi_{\mathsf{k}_{j,A}}(t_j)$ denotes the jth pseudorandom shared value from $\mathbb{Z}_q$. Define the associated element in R_q by the polynomial $\sum s_j X^j$.

If the underlying PRF family has range $[-T, \ldots, T]$ over $\mathbb{Z}_q$ then the output of the above PRSS is an element in R_q whose coefficients lie in the range $[-\binom{n}{t}T, \binom{n}{t}T]$. To ease notation we write $\mathbf{s} = \sum_{A\subset\mathcal{P}:|A|=n-t, P_i\in A} \psi_{\mathbf{k}_A}(\mathbf{t})$ for the shared value in R_q, and $[\mathbf{s}]_i = \sum_{A\subset\mathcal{P}:|A|=n-t, P_i\in A} \psi_{\mathbf{k}_A}(\mathbf{t})\cdot f_A(i)$ for the shares themselves. We note that in general $\binom{n}{t}$ becomes exponentially large, specially if t is a constant fraction of n; however in most practical applications of threshold cryptography, the number of parties n is indeed expected to be small.

Canonical Embedding Norm: Here we recall some results on cyclotomic fields that we need to estimate the parameters of our protocol instantiations. For details regarding properties of canonical norms we refer to [31, 30, 25]. Given a polynomial $a \in R$ we denote by $\|\mathbf{a}\|_\infty = \max_{0\le i\le N-1} |a_i|$ the standard l_∞-norm. All estimates of noise are taken with respect to the *canonical embedding norm* $\|a\|_\infty^{\mathsf{can}} = \|\sigma(a)\|_\infty$, where σ is the canonical embedding $R \to \mathbb{C}^{\phi(m)}$ defined by $\sigma : a \mapsto a(\zeta_m^k)$, $k \in \mathbb{Z}_m^*$ and ζ_m a fixed primitive mth root of unity. When $a \in R_q$, for some modulus q, we need the canonical embedding norm reduced modulo q:

$$|a|_q^{\mathsf{can}} = \min\{\|a'\|_\infty^{\mathsf{can}} : a' \in R \text{ and } a' \equiv a \pmod q\}.$$

To map from norms in the canonical embedding to norms on the coefficients of the polynomials defining the elements in R we note that we have $\|\mathbf{a}\|_\infty \le c_m \cdot \|a\|_\infty^{\mathsf{can}}$, where c_m is the *ring constant*. Since we fix the choice of our base field polynomial as a 2^kth cyclotomic polynomial, we have $c_m = 1$.

7.2 The Basic L-levelled Packed BGV-type Cryptosystem

We review the BGV L-levelled Packed SHE scheme. The scheme is parametrized by a security parameter κ, for a fixed number of levels $L+1$. Note, we use $L+1$ levels in our scheme description to make the presentation consistent with the abstract scheme from Section 3. For $\mathfrak{l} = 0, \ldots, L$, fix a chain of moduli $q_\mathfrak{l} = \prod_{i=0}^{\mathfrak{l}} p_i$, with p_i a prime number. Encryption generates level L ciphertexts with respect to the largest modulus q_L. In the $\mathfrak{l}$th level of the scheme ciphertexts consist of two elements in $R_{q_\mathfrak{l}}$, $\mathfrak{l} = 0, \ldots, L$. Throughout homomorphic evaluation we will force a *universal* bound B on the noise contained in ciphertexts (when measured in the canonical embedding norm reduced modulo q) after a SHE.LowerLevel execution. Since $\|\mathbf{a}\|_\infty \le \|a\|_\infty^{\mathsf{can}} \le B$ this provides an upper bound also on the coefficients used in the underlying decryption algorithm, for such outputs of SHE.LowerLevel. For a description of the algorithm SHE.LowerLevel see [31]; where it is called *modulus switching*.

However, when applying decryption, or distributed decryption, we will apply the procedure to a ciphertext which is not the direct output of a SHE.LowerLevel operation. In particular we assume that the canonical norm of the noise of an element passed to the decryption procedure will be bounded by B_{dec}. The decryption procedures will then return the correct output if we have $B_{\mathsf{dec}} \le q_0/2$. For distributed decryption we will need to "boost" this bound to $2^{\mathsf{exp}} \cdot B_{\mathsf{dec}}$, where exp is a "closeness parameter" relating to the statistical security parameter sec. Thus distributed decryption will be work if and only if $2^{\mathsf{exp}} \cdot B_{\mathsf{dec}} < q_0/2$. Below we specify the basic algorithms for the BGV scheme; we will then discuss the extensions to cope with the full syntax of our scheme in Definition 2.

Before presenting the methods we need to pause briefly to remind the reader about modulus switching: A ciphertext at level $\mathfrak{l}$ is given by a pair $\mathsf{c} = (c_0, c_1) \in R_{q_\mathfrak{l}}^2$ and the decryption procedure computes, for the global secret key $\mathsf{sk} \in R$,

$$[c_0 - \mathsf{sk}\cdot c_1]_{q_\mathfrak{l}} = c_0 - \mathsf{sk}\cdot c_1 \pmod{q_\mathfrak{l}}$$

where we take the symmetric modular operation in the range $[-q_\mathfrak{l}/2, \ldots, q_\mathfrak{l}/2]$. The value $[c_0 - \mathsf{sk} \cdot c_1]_{q_\mathfrak{l}}$ can be interpreted as an element in R, and the associated noise value of the ciphertext is the canonical norm of this element. After each homomorphic operation the norm of the noise in the ciphertexts increases. To reduce it the modulus switching technique ([11, 10]) is used. This procedure takes as input a ciphertext $\mathfrak{c} = (c_0, c_1) \in R^2_{q_\mathfrak{l}}$, with estimated noise ν and transforms it into a ciphertext $\mathfrak{c}' \in R^2_{q_{\mathfrak{l}'}}$ at level $\mathfrak{l}'$, with noise magnitude ν', by scaling down $\mathfrak{c}$ by a factor $q_{\mathfrak{l}'}/q_\mathfrak{l}$ and then rounding to get back an integer ciphertext. The ciphertext $\mathfrak{c}' = (c'_0, c'_1)$ satisfies $[c_0 - \mathsf{sk} \cdot c_1]_{q_\mathfrak{l}} \equiv [c'_0 - \mathsf{sk} \cdot c'_1]_{q_{\mathfrak{l}'}} \mod p$ and $\nu' < \nu$. This modulus switching operation corresponds to our operation SHE.LowerLevel from Definition 2.

1. $\mathsf{SHE.KeyGen}(1^\kappa) \to (\mathsf{pk}, \mathsf{ek}, \mathsf{sk})$: Outputs a secret key $\mathsf{sk} \leftarrow \mathcal{HWT}(h, N)$, a common public key $\mathsf{pk} = (a, b)$ such that $a \leftarrow \mathcal{U}_s(q_L, N)$ and $b = a \cdot \mathsf{sk} + p \cdot e$, with $e \leftarrow \mathcal{DG}(\sigma^2, N)$. This algorithm also outputs the evaluation key ek which consists of $N + 1$ public "key-switching matrices" $W_{\mathsf{sk}^2 \to \mathsf{sk}}$ and $W_{\sigma_i(\mathsf{sk}) \to \mathsf{sk}}$ and $\sigma_i \in \mathsf{Gal}$ for $i = 1, \ldots, N$. See [31] for how these are defined.
2. $\mathsf{SHE.Enc}_{\mathsf{pk}}(\mathbf{m}) \to (\mathfrak{c}, L)$: Given a plaintext $\mathbf{m} \in R_p$, the encryption algorithm samples $(v, e_0, e_1) \leftarrow \mathcal{RC}_s(0.5, \sigma^2, N)$ and then computes in R_{q_L},
$$c_0 = b \cdot v + p \cdot e_0 + \mathbf{m} \quad \text{and} \quad c_1 = a \cdot v + p \cdot e_1.$$
3. $\mathsf{SHE.Dec}_{\mathsf{sk}}(\mathfrak{c}, \mathfrak{l}) \to \mathbf{m}'$: Note, this algorithm is never called in our scheme, we just present it here so as to define correctness and to define what we mean by a message associated to a ciphertext. The algorithm takes as input a ciphertext $\mathfrak{c} = (c_0, c_1) \in R^2_{q_\mathfrak{l}}$ and outputs a plaintext $\mathbf{m}' \in R_p$. This algorithm uses the secret key sk to compute
$$\mu = c_0 - \mathsf{sk} \cdot c_1 = \mathbf{m}' + p \cdot (e \cdot v + e_0 - s \cdot e_1) = \mathbf{m}' + p \cdot u$$
in $R_{q_\mathfrak{l}}$ and then obtains $\mathbf{m}' = (\mu \mod p)$. We denote by ν the estimated noise magnitude obtained by using the canonical embedding norm and we require that $\nu < B_{\mathsf{dec}}$. This decryption procedure will correctly work if $B_{\mathsf{dec}} < q_\mathfrak{l}/2$.
4. $\mathsf{SHE.Eval}_{\mathsf{ek}}(C^{\mathsf{sub}}, (\mathfrak{c}_1, \mathfrak{l}_1^{in}), \ldots, (\mathfrak{c}_{\ell_{in}}, \mathfrak{l}_{\ell_{in}}^{in})) \to (\hat{\mathfrak{c}}_1, \mathfrak{l}_1^{out}), \ldots, (\hat{\mathfrak{c}}_{\ell_{out}}, \mathfrak{l}_{\ell_{out}}^{out})$: This consists of three separate algorithm SHE.Add, SHE.Mult and SHE.ScalarMult for homomorphically evaluating addition and multiplication gates.
   - $\mathsf{SHE.Add}_{\mathsf{ek}}((\mathfrak{c}_1, \mathfrak{l}_1), (\mathfrak{c}_2, \mathfrak{l}_2))$: It produces a ciphertext $\mathfrak{c}_{\mathsf{Add}}$ in $R^2_{q_\mathfrak{l}}$, with $\mathfrak{l} = \min\{\mathfrak{l}_1, \mathfrak{l}_2\}$. This is performed by first applying $\mathfrak{c}'_i = \mathsf{SHE.LowerLevel}_{\mathsf{ek}}((\mathfrak{c}_i, \mathfrak{l}_i), \mathfrak{l})$ and then taking the coordinate-wise addition of $\mathfrak{c}'_1$ and $\mathfrak{c}'_2$. The noise magnitude of the resulting ciphertext is at most the sum of the noise in $\mathfrak{c}_1$ and $\mathfrak{c}_2$.
   - $\mathsf{SHE.Mult}_{\mathsf{ek}}((\mathfrak{c}_1, \mathfrak{l}_1), (\mathfrak{c}_2, \mathfrak{l}_2))$: This produces a ciphertext $\mathfrak{c}_{\mathsf{Mult}}$ in $R^2_{q_\mathfrak{l}}$, with $\mathfrak{l} = \min\{\mathfrak{l}_1, \mathfrak{l}_2\} - 1$. This is done in one of two ways (so as to minimize the overall parameter sizes in our scheme).
     - If $\mathfrak{l} \neq 1$ then one first applies $\mathfrak{c}'_i = \mathsf{SHE.LowerLevel}_{\mathsf{ek}}((\mathfrak{c}_i, \mathfrak{l}_i), \mathfrak{l})$, then the resulting ciphertexts are tensored. This results in a ciphertext $\tilde{\mathfrak{c}}$ is a vector of higher dimension ([12]) and corresponding to a valid ciphertext of the SIMD-product of the associated plaintexts $\mathbf{m}_1 \cdot \mathbf{m}_2$ with respect to a secret key sk' that is the tensor product of the secret key sk with itself. The Key Switching procedure ([31]) is then applied, using the matrix $W_{\mathsf{sk}^2 \to \mathsf{sk}}$, to obtain a valid ciphertext $\mathfrak{c}_{\mathsf{Mult}} \in R^2_{q_\mathfrak{l}}$ with respect to the original secret key sk. The noise magnitude in $\mathfrak{c}_{\mathsf{Mult}}$ is at approximately product of norms of the noise in $\mathfrak{c}'_1$ and $\mathfrak{c}'_2$.
     - If $\mathfrak{l} = 1$ then one applies the tensor operation to $\mathfrak{c}_1$ and $\mathfrak{c}_2$ directly, then the key switching is performed and only then is a SHE.LowerLevel operation performed. This results in us needing a larger prime p_1 than one would otherwise need, but more importantly a smaller p_0.
   - $\mathsf{SHE.ScalarMult}_{\mathsf{ek}}((\mathfrak{c}, \mathfrak{l}), \mathbf{a})$: If $\mathfrak{c} = (c_0, c_1)$ then one can obtain a homomorphic scalar multiplication by evaluating $\mathfrak{c}' = (\mathbf{a} \cdot c_0, \mathbf{a} \cdot c_1)$. This procedure increases the noise, but not by as much as a normal multiplication. Therefore we shall ignore the noise increase produced by scalar multiplication in our analysis.

Using the evaluation key we can also define an addition homomorphic operation as in [30, 31],

- $\mathsf{SHE.Permute}_{\mathsf{ek}}((\mathfrak{c}, \mathfrak{l}), \sigma) \to (\hat{\mathfrak{c}}_{\mathsf{Permute}}, \mathfrak{l})$: Given $\sigma \in \mathsf{Gal}$ and a ciphertext $\mathfrak{c} = (c_0, c_1) \in R^2_{q_\mathfrak{l}}$, corresponding to a plaintext $\mathbf{m} \in R_p$, this generates a ciphertext $\hat{\mathfrak{c}}_{\mathsf{Permute}} = (\hat{c}_0, \hat{c}_1) \in R^2_{q_\mathfrak{l}}$ corresponding to $\sigma(m)$, with respect to the secret key $\sigma(\mathsf{sk})$. Key switching is then applied, using the keyswitching matrix $W_{\sigma(\mathsf{sk}) \to \mathsf{sk}}$ to produce a ciphertext, $\hat{\mathfrak{c}}_{\mathsf{Permute}}$ decryptable under sk.

7.3 Defining SHE.Pack and SHE.Unpack for BGV

Despite our scheme being a packed SHE scheme it can still evaluate unpacked ciphertexts; indeed many of the instances of packed SHE schemes were originally conceived in the unpacked case by taking the map χ to be $\chi(m) = (m, m, \ldots, m)$, i.e. the *diagonal embedding*. For example this is the case with the schemes in [39, 12, 11, 9] etc all of which have packed counterparts. However, such a choice of χ is not efficient if one is interested in packing and unpacking encryptions of elements in $\mathbb{F}_p$. We wish to define two functions $\mathsf{SHE.Pack}$ and $\mathsf{SHE.Unpack}$; the first of which takes N ciphertexts $\mathfrak{c}_i$ at level $\mathfrak{l}_i$ with the associated plaintext vector $\chi(m_i)$ for $m_i \in \mathbb{F}_p$, and produces a single ciphertext $\mathfrak{c}$ at level $\min(\mathfrak{l}_i)$ with the associated plaintext vector $\mathbf{m} = (m_1, \ldots, m_N) \in \mathcal{M}$. The second function performs the reverse operation.

In what follows we let $\mathbf{e}_i$ denote the i-th unit vector in $\mathcal{M}$, i.e. the element which is zero except for a one in the i-th position. To ease notation we let $\oplus$ and $\otimes$ denote the operations of applying the $\mathsf{SHE.Add}$ and $\mathsf{SHE.Mult}$/$\mathsf{SHE.ScalarMult}$ operations respectively, we also let $\sigma(\mathfrak{c})$ denote applying the $\mathsf{SHE.Permute}$ operation to a ciphertext $\mathfrak{c}$ and map $\sigma \in \mathsf{Gal}$. If we define χ by the diagonal embedding then $\mathsf{SHE.Pack}$ can be defined in the following way

$$\mathsf{SHE.Pack}(\mathfrak{c}_1, \ldots, \mathfrak{c}_N) = \bigoplus_{i=1}^{N} \mathbf{e}_i \otimes \mathfrak{c}_i,$$

i.e. $\mathsf{SHE.Pack}$ is an $O(N)$ operation. However, $\mathsf{SHE.Unpack}$ needs to be performed as follows for $i = 1, \ldots, N$,

$$\mathsf{SHE.Unpack}(\mathfrak{c}) = \left(\bigoplus_{j=1}^{N} \sigma_{1\to j}(\mathbf{e}_1 \otimes \mathfrak{c}), \ldots, \bigoplus_{j=1}^{N} \sigma_{N\to j}(\mathbf{e}_N \otimes \mathfrak{c}) \right)$$

i.e. $\mathsf{SHE.Unpack}$ is an $\mathcal{O}(N^2)$ operation. On the other hand, if we define χ to be the map $\chi(m) = (m, 0, \ldots, 0)$ then we can define $\mathsf{SHE.Pack}$ and $\mathsf{SHE.Unpack}$ by the following $\mathcal{O}(N)$ operations;

$$\mathsf{SHE.Pack}(\mathfrak{c}_1, \ldots, \mathfrak{c}_N) = \bigoplus_{i=1}^{N} \sigma_{i\to j}(\mathfrak{c}_i), \quad \mathsf{SHE.Unpack}(\mathfrak{c}) = (\mathbf{e}_1 \otimes \mathfrak{c}, \sigma_{2\to 1}(\mathbf{e}_2 \otimes \mathfrak{c}), \ldots, \sigma_{N\to 1}(\mathbf{e}_N \otimes \mathfrak{c})).$$

Thus we will utilize the mapping $\chi(m) = (m, 0, \ldots, 0)$ in our proposal.

7.4 Distributed Decryption Protocol

All that remains to define our Threshold L-levelled Packed SHE system based on BGV is to present the distributed decryption protocol. Note that we do not use the key-homomorphic properties of RLWE schemes as previously used in [1, 35, 2]. Instead, we follow the approach of [6], where the authors construct a threshold variant of Regev's cryptosystem ([38]); we adapt this method to our situation.

At a high level the method works as follows: we modify the $\mathsf{SHE.KeyGen}$ algorithm so that it also outputs for each party P_i a key dk_i for performing distributed decryption. The key dk_i consists of two components; i.e. $\mathsf{dk}_i = (\mathsf{sk}_i, \mathbf{k}_i)$. The values sk_i form a Shamir sharing over the ring R_{q_0} of the secret key sk, with threshold t. The value $\mathbf{k}_i$ are the associated keys for the PRSS described above. Given a common ciphertext $\mathfrak{c} = (c_0, c_1) \in R_{q_\mathfrak{l}}$ as input (for decryption), the parties first apply $\mathsf{SHE.LowerLevel}$ to reduce the ciphertext to level zero. Then each party P_i computes a decryption share $\bar{\mu}_i$ using his private sk_i and a PRSS over R_{q_0} as described earlier. The underlying PRF we assume produces values in the range

$$\left[-\frac{(2^{\mathsf{exp}} - 1) \cdot B_{\mathsf{dec}}}{p \cdot \binom{n}{t}}, \frac{(2^{\mathsf{exp}} - 1) \cdot B_{\mathsf{dec}}}{p \cdot \binom{n}{t}} \right],$$

where B_{dec} is the bound on the canonical norm of an element being decrypted mentioned earlier, and hence an upper bound on the size of the coefficients of the noise polynomial reconstructed during the standard decryption procedure. See Section 8 for a detailed discussion of B_{dec}. The choice of this range of the underlying PRF family means that the values output by the PRSS will be shares of elements in $R_{q_\mathfrak{l}}$ whose coefficients lie in the range $[-(2^{\mathsf{exp}} - 1) \cdot$

$B_{\mathsf{dec}}/p, (2^{\mathsf{exp}} - 1) \cdot B_{\mathsf{dec}}/p]$. Note, that $\binom{n}{t}$ for $t \approx n/2$ grows very fast, and so the for the above range of the PRF to be suitably large we require that n is small. In our discussion we implicitly assume n to be small, say $n < 10$.

Recall distributed decryption is defined by two algorithms SHE.ShareDec and SHE.ShareCombine. These are defined by the following procedures:

- $\mathsf{SHE.ShareDec}_{\mathsf{dk}_i}((\mathfrak{c}, \mathfrak{l})) \rightarrow \bar{\mu}_i$:
  1. Apply $\mathsf{SHE.LowerLevel}_{\mathsf{ek}}((\mathfrak{c}, \mathfrak{l}), 0)$ to obtain the ciphertext (c_0, c_1) at level zero (unless $\mathfrak{c}$ is already at level zero).
  2. Compute $\mu_i = [\mu]_i = [c_0 - \mathsf{sk} \cdot c_1]_i = c_0 - \mathsf{sk}_i \cdot c_1$ where the computation is in R_{q_0}.
  3. Execute the PRSS, using the PRF keys $\mathbf{k}_i$, to obtain a Shamir's share r_i of a "random" value $r \in R_{q_0}$ such that $r = \sum_{\mathbf{k}_A} \psi_{\mathbf{k}_A}(\mathbf{t})$ and $\|r\|_\infty < ((2^{\mathsf{exp}} - 1) \cdot B_{\mathsf{dec}})/p$, for some agreed vector of values $\mathbf{t}$ which are a function of the input ciphertext $\mathfrak{c}$.
  4. Compute $\bar{\mu}_i = [\bar{\mu}]_i = [\mu + p \cdot r]_i = \mu_i + p \cdot r_i$ and output $\bar{\mu}_i$ as the decryption share.
- $\mathsf{SHE.ShareCombine}((\mathfrak{c}, \mathfrak{l}), \{\bar{\mu}_i\}_{i \in [1,\ldots,n]}) \rightarrow \mathbf{m}'$: given a set n of decryption shares $\bar{\mu}_i$ and (in the malicious setting) an error correction procedure, reconstruct $\bar{\mu} = \mu + p \cdot r$ by applying the error correction procedure to $\{\bar{\mu}_i\}_{i \in [1\ldots,n]}$ and output $\mathbf{m}' = (\bar{\mu} \mod p)$.

Note decryption will work as long as the reconstructed value $\bar{\mu}$ is less than $q_0/2$, i.e. we require $2^{\mathsf{exp}} \cdot B_{\mathsf{dec}} < q_0/2$ (see the next section for details).

We pause to note the different situations where one obtains correct message recovery from SHE.ShareCombine. In the case of passive adversaries we will show (in the next section) that the above distributed decryption procedure is secure as long as $t < n$. Since we are using Shamir sharing, in the presence of $t < n/3$ active corruptions, using the natural error correction properties (namely Reed-Solomon (RS) error correction), we can correctly recover the message at the end of SHE.ShareCombine.

When $t < n/2$ a little more work is involved; if an adversary sends an incorrect share then this can be detected, again because we are using Shamir as the underlying secret sharing scheme. At this point the parties execute a party elimination strategy in which they require each other to prove in zero-knowledge that the provided share is correct. Once the cheater party(s) have been determined they are eliminated from the protocol and the protocol resumes. Thus for active adversaries and $t < n/2$ we may require a grand total of an extra $n^2 \cdot t$ zero-knowledge proofs to be constructed, irrespective of the size of the circuit in our main protocol; see Section 6 for more details.

7.5 Security of Our Threshold BGV Instantiation

Recall from earlier we require four security properties:

- Key Generation Security.
- Semantic Security.
- Correct Share Decryption.
- Share Simulation Indistinguishability.

We now discuss each of these in turn.

Key Generation Security: The required properties of the keys produced by the key generation algorithm follow from the security properties of the Shamir secret sharing scheme used to share sk. We note in our main protocol we assume an ideal functionality to distribute such keys, and so there is no "Key Generation" protocol to analyse.

Semantic Security: The follows from the standard semantic security of the BGV scheme. However, we need to deal with the fact that the adversary has access to shares of the underlying secret key and the keys to the PRSS. A standard simulation shows that security in our setting reduces to that in the standard setting.

Correct Share Decryption: The infinity norm of the element $\bar{\mu} = \mu + p \cdot r$ produced by the algorithm SHE.ShareCombine is bounded by $B_{\mathsf{dec}} + p \cdot (2^{\mathsf{exp}} - 1) \cdot B_{\mathsf{dec}}/p \approx 2^{\mathsf{exp}} \cdot B_{\mathsf{dec}}$. If $2^{\mathsf{exp}} \cdot B_{\mathsf{dec}} < q_0/2$ then correct decryption will result.

Share Simulation Indistinguishability: We need to present a PPT algorithm (simulator) SHE.ShareSim which when given a ciphertext $\mathfrak{c} \in R_{q_\mathfrak{l}}^2$ with associated plaintext $\mathbf{m} \in R_p$, a subset $I \subset \{1, \ldots, n\}$ such that $|I| = t$, and a set

of t decryption shares $\{\bar{\mu}_i\}_{i\in I}$, where $\bar{\mu}_i = \mathsf{SHE.ShareDec}_{\mathsf{dk}_i}((\mathfrak{c},\mathfrak{l}))$, can simulate the remaining $(n-t)$ decryption shares $\{\bar{\mu}_j^*\}$ in such a way that the following two following distributions are statistically indistinguishable:

$$\left(\{\bar{\mu}_i\}_{i\in I}, \{\bar{\mu}_j^*\}_{j\in\bar{I}}\right) \overset{s}{\approx} \left(\{\bar{\mu}_i\}_{i\in I}, \{\bar{\mu}_j\}_{j\in\bar{I}}\right),$$

where $\bar{I} = \{1,\ldots,n\}\setminus I$. i.e. one cannot distinguish the real shares for the set $\bar{I}$ (as computed by $\mathsf{SHE.ShareDec}$ algorithm) with ones produced by the simulator. Moreover, we require the statistical distance between the two distributions to be bounded by $2^{-\mathsf{sec}}$. The simulator is constructed as follows:

1. Let $\mathbf{k}_A^{(I)}$ denote the set of keys for the PRSS that have been given to parties P_i where $i \in I$, and let $\mathbf{k}_A^{(\bar{I})}$ denote the set of keys for the PRSS held by P_j, for $j \in \bar{I}$.
2. The simulator first computes
$$r' = \sum_{\mathbf{k}\in\mathbf{k}_A^{(I)}} \psi_{\mathbf{k}}(\mathbf{t}) + \sum_{\mathbf{k}\in\mathbf{k}_A^{(\bar{I})}} r_{\mathbf{k}},$$
where each $r_{\mathbf{k}} \in R_{q_\mathfrak{l}}$ is chosen such that
$$\|r_{\mathbf{k}}\|_\infty < \frac{(2^{\mathsf{exp}}-1)\cdot B_{\mathsf{dec}}}{p\cdot\binom{n}{t}}.$$
In this way $\|r'\|_\infty < \frac{(2^{\mathsf{exp}}-1)\cdot B_{\mathsf{dec}}}{p}$.
3. Let $\bar{\mu}^* = \mathbf{m} + p\cdot r'$. For each $j \in \bar{I}$, the simulator outputs $\bar{\mu}_j^*$ such that $\left(\{\bar{\mu}_j^*\}_{j\in\bar{I}}, \{\bar{\mu}_i\}_{i\in I}\right)$ is a consistent vector of shares of $\bar{\mu}^*$; *i.e.* the simulator deterministically computes consistent shares for the honest parties via Lagrange interpolation of the $t+1$ values, $\bar{\mu}^*$ and $\{\bar{\mu}_i\}_{i\in I}$.

Before proving the properties of the simulation, we recall the following lemma from [2]:

Lemma 1 (Smudging Lemma [2]). *Let B_1 and B_2 be positive integers and let $e_1 \in [-B_1, B_1]$ be a fixed integer and let $e_2 \in [-B_2, B_2]$ be chosen uniformly and randomly. Then the statistical distance between the distribution of e_2 and $e_2 + e_1$ is B_1/B_2.*

To prove the properties of the simulation, we first note that similar to the last stage of the simulation above, the real shares for the honest parties can be constructed (deterministically) from $\bar{\mu}$ and the shares held by the t dishonest parties. Thus, to prove indistinguishability of the real and simulated shares, it suffices to prove that $\bar{\mu}^* = \mathbf{m} + p\cdot r'$ and $\bar{\mu} = \mu + p\cdot r$ are statistically close[4]. To see this is indeed the case, we first note that $\mu + p\cdot r$ and $\mu + p\cdot r'$ are indistinguishable (by construction) and that r' is uniform in an exponentially larger range than μ (recall that $\|\mu\|_\infty < B_{\mathsf{dec}}$ and $\|r'\|_\infty < \frac{(2^{\mathsf{exp}}-1)\cdot B_{\mathsf{dec}}}{p}$). By application of the Smudging lemma, the statistical distance between the distribution of $\mu + p\cdot r'$ and the uniform distribution of polynomials with coefficients in $[-(2^{\mathsf{exp}}-1)\cdot B_{\mathsf{dec}}, (2^{\mathsf{exp}}-1)\cdot B_{\mathsf{dec}}]$ is exactly $N/(2^{\mathsf{exp}}-1)$.

To conclude the proof, we next claim that the distribution of $\mathbf{m} + p\cdot r'$ is statistically indistinguishable from the uniform distribution of polynomials with coefficients $[-(2^{\mathsf{exp}}-1)\cdot B_{\mathsf{dec}}, (2^{\mathsf{exp}}-1)\cdot B_{\mathsf{dec}}]$. This follows from the fact that the statistical distance between the two distributions is $\frac{N\cdot p}{B_{\mathsf{dec}}\cdot(2^{\mathsf{exp}}-1)}$ (which itself follows from the Smudging Lemma and the fact that $\mathbf{m} \in R_p$). It follows from the triangle inequality that the overall statistical distance between the distribution of $\bar{\mu}^* = \mathbf{m} + p\cdot r'$ and $\bar{\mu} = \mu + p\cdot r$ is upper bounded by $\frac{N\cdot(p+B_{\mathsf{dec}})}{B_{\mathsf{dec}}\cdot(2^{\mathsf{exp}}-1)}$. Choose

$$\mathsf{exp} = \mathsf{sec} + \max\left(\log_2\left(\frac{N\cdot(p+B_{\mathsf{dec}})}{B_{\mathsf{dec}}}\right), \log_2(N)\right).$$

Since $p < B_{\mathsf{dec}}$ this simplifies to $\mathsf{exp} = \mathsf{sec} + \log_2(N) + 1$, and we can therefore ensure the statistical distance is bounded by $2^{-\mathsf{sec}}$ which can be made arbitrarily small by our choice of exp.

[4] For statistically close distributions $X \overset{s}{\approx} Y$ and any deterministic procedure A applied to those distributions it is the case that $A(X) \overset{s}{\approx} A(Y)$.

7.6 Batch Distributed Decryption

Using a well known technique presented in [4, 24], we can perform a batch of $t+1 = \Theta(n)$ distributed decryption, and hence evaluate a batch of $t+1$ Refresh gates at the communication cost of performing two distributed decryptions. The following technique applies to our main MPC protocol if the batch of refresh gates are *independent*, meaning the output wire of one does not lead to the input of the other.

Given a value shared among the parties, its public reconstruction requires each party to send the share (of the value) it holds to every other party. This requires $n \cdot (n-1)$ pair-wise communication of shares. So for $t+1$ shared values, the public reconstruction will require $\mathcal{O}(n^3)$ pair-wise communication of shares. In what follows, it is shown how the above can be achieved with the same cost of public reconstruction of a single value, namely with a communication of $2 \cdot n \cdot (n-1) = \mathcal{O}(n^2)$ shares. The idea was used in the information theoretically secure MPC protocols of [4, 24].

Let $u^{(1)}, \ldots, u^{(t+1)}$ be $t+1$ shared values. First the $t+1$ shared values are "expanded" to n shared values, say $v^{(1)}, \ldots, v^{(n)}$ by applying a linear function locally. Specifically, if the underlying LSS is Shamir, then we can interpret $u^{(1)}, \ldots, u^{(t+1)}$ as the coefficients of a polynomial of degree at most t, say $u(\cdot)$ and let $v^{(1)}, \ldots, v^{(n)}$ be the n distinct points on this polynomial. Now notice that obtaining $v^{(1)}, \ldots, v^{(n)}$ from $u^{(1)}, \ldots, u^{(t+1)}$ is a linear function and by (locally) applying the same linear function on the sharings of $u^{(1)}, \ldots, u^{(t+1)}$, the parties can obtain sharings of $v^{(1)}, \ldots, v^{(n)}$. Now each $v^{(i)}$ is reconstructed only to P_i and this costs $\mathcal{O}(n^2)$ communication of shares. Finally every P_i sends $v^{(i)}$ to every other party (which costs another $\mathcal{O}(n^2)$ communication) and then every party can reconstruct $u(\cdot)$ and hence $u^{(1)}, \ldots, u^{(t+1)}$.

In our setting all of the above sharing is done using Shamir over the ring R_{q_0}. It is easy to see that the above can be carried out with no change to the underlying SHE scheme. Thus assuming our initial circuit is large enough, i.e. there are enough independent Refresh gates at each level, we can obtain a performance improvement of $(t+1)/2$.

8 Parameter Calculation

In [31] a concrete set of parameters for the BGV SHE scheme was given for the case of binary message spaces, and arbitrary L. In [22] this was adapted to the case of message space R_p for 2-power cyclotomic rings, but only for the schemes which could support one level of multiplication gates (i.e. for $L = 1$). In this section we combine these analyses to produce parameter estimations for the case we require of arbitrary L and messages defined by a "large prime", e.g. $p \approx 2^{32}, 2^{64}$ or 2^{128}. We assume in this section that the reader is familiar with the analysis and algorithms from [31]; we mainly point out the differences in estimates for our case.

Our analysis will make extensive use of the following fact: If $a \in R$ be chosen from a distribution such that the coefficients are distributed with mean zero and standard deviation σ, then if ζ_m is a primitive mth root of unity, we can use $6 \cdot \sigma$ to bound $a(\zeta_m)$ and hence the canonical embedding norm of a. If we have two elements with variances σ_1^2 and σ_2^2, then we can bound the canonical norm of their product with $16 \cdot \sigma_1 \cdot \sigma_2$.

Recall from Section 7 that we require a chain of moduli $q_0 < q_1 \ldots < q_L$ corresponding to each level of the scheme, where $q_L = q_0 \cdot \prod_{i=1}^{i=L} p_i$. Note that we evaluate a depth L circuit from a chain of $L+1$ moduli. Also note, that we apply a SHE.LowerLevel (a.k.a. modulus switch) algorithm *before* a multiplication operation, except when multiplying at level one. This often leads to lower noise values in practice (which a practical instantiation can make use of). In addition it eliminates the need to perform a modulus switch after encryption.

We utilize the following constants described in [22], which are worked out for the case of message space defined modulo p (the constants in [22] make use of an additional parameter n, arising from the key generation procedure. In our case we can take this constant equal to one).

$$
\begin{aligned}
B_{\mathsf{Clean}} &= N \cdot p/2 + p \cdot \sigma \cdot \left(\frac{16 \cdot N}{\sqrt{2}} + 6 \cdot \sqrt{N} + 16 \cdot \sqrt{h \cdot N} \right) \\
B_{\mathsf{Scale}} &= p \cdot \sqrt{3 \cdot N} \cdot \left(1 + \frac{8}{3} \cdot \sqrt{h} \right) \\
B_{\mathsf{Ks}} &= p \cdot \sigma \cdot N \cdot \left(1.49 \cdot \sqrt{h \cdot N} + 2.11 \cdot h + 5.54 \cdot \sqrt{h} + 1.96\sqrt{N} + 4.62 \right)
\end{aligned}
$$

The constants are used in the following manner: A freshly encrypted ciphertext at level L has noise bounded by B_{Clean}. In the worst case, when applying $\mathsf{SHE.LowerLevel}$ to a ciphertext at level $\mathfrak{l}$ with noise bounded by B' one obtains a new ciphertext at level $\mathfrak{l}-1$ with noise bounded by

$$\frac{B'}{p_{\mathfrak{l}}} + B_{\mathsf{Scale}}.$$

When applying the tensor product multiplication operation to ciphertexts of a given level $\mathfrak{l}$ of noise B_1 and B_2 one obtains a new ciphertext with noise given by

$$B_1 \cdot B_2 + \frac{B_{\mathsf{Ks}} \cdot q_{\mathfrak{l}}}{P} + B_{\mathsf{Scale}},$$

where P is a value to be determined later. As in [31] we define a small "wiggle room" ξ which we set to be equal to eight; this is set to enable a number of additions to be performed without needing to individually account for them in our analysis.

A general evaluation procedure begins with a freshly encrypted ciphertext at level L with noise B_{Clean}. When entering the first multiplication operation we first apply a $\mathsf{SHE.LowerLevel}$ operation to reduce the noise to our universal bound B. We therefore require

$$\frac{\xi \cdot B_{\mathsf{Clean}}}{p_L} + B_{\mathsf{Scale}} \leq B,$$

i.e.

$$p_L \geq \frac{8 \cdot B_{\mathsf{Clean}}}{B - B_{\mathsf{Scale}}}. \tag{1}$$

We now turn to dealing with the $\mathsf{SHE.LowerLevel}$ operation which occurs before a multiplication gate at level $\mathfrak{l} \in [2, \ldots, L-1]$. We perform a worst case analysis and assume that the input ciphertexts are at level $\mathfrak{l}-1$. We can then assume that the input to the tensoring operation in the previous multiplication gate (just after the previous $\mathsf{SHE.LowerLevel}$) was bounded by B, and so the output noise from the previous multiplication gate for each input ciphertext is bounded by $B^2 + B_{\mathsf{Ks}} \cdot q_{\mathfrak{l}}/P + B_{\mathsf{Scale}}$. This means the noise on entering the $\mathsf{SHE.LowerLevel}$ operation is bounded by ξ times this value, and so to maintain our invariant we require

$$\frac{\xi \cdot B^2 + \xi \cdot B_{\mathsf{Scale}}}{p_{\mathfrak{l}}} + \frac{\xi \cdot B_{\mathsf{Ks}} \cdot q_{\mathfrak{l}}}{P \cdot p_{\mathfrak{l}}} + B_{\mathsf{Scale}} \leq B.$$

Rearranging this into a quadratic equation in B we have

$$\frac{\xi}{p_{\mathfrak{l}}} \cdot B^2 - B + \left(\frac{\xi \cdot B_{\mathsf{Scale}}}{p_{\mathfrak{l}}} + \frac{\xi \cdot B_{\mathsf{Ks}} \cdot q_{\mathfrak{l}-1}}{P} + B_{\mathsf{Scale}} \right) \leq 0.$$

We denote the constant term in this equation by $R_{\mathfrak{l}-1}$. We now assume that all primes $p_{\mathfrak{l}}$ are of roughly the same size, and noting the we need to only satisfy the inequality for the largest modulus $\mathfrak{l} = L-1$. We now fix R_{L-2} by trying to ensure that R_{L-2} is close to $B_{\mathsf{Scale}} \cdot (1 + \xi/p_{L-1}) \approx B_{\mathsf{Scale}}$, so we set $R_{L-2} = (1 - 2^{-3}) \cdot B_{\mathsf{Scale}}(1 + \xi/p_{L-1})$, and obtain

$$P \approx 8 \cdot \frac{\xi \cdot B_{\mathsf{Ks}} \cdot q_{L-2}}{B_{\mathsf{Scale}}}, \tag{2}$$

since $B_{\mathsf{Scale}} \cdot (1 + \xi/p_{L-1}) \approx B_{\mathsf{Scale}}$.

To ensure we have a solution we require $1 - 4 \cdot \xi \cdot R_{L-2}/p_{L-1} \geq 0$, which implies we should take, for $i = 2, \ldots, L-1$,

$$p_i \approx 4 \cdot \xi \cdot R_{L-2} \approx 32 \cdot B_{\mathsf{Scale}}. \tag{3}$$

Recall that the final multiplication is executed in a different manner. We do not modulus switch before the multiplication, but afterwards. We analyse the implication of this, for the size of p_1, from the point of view of our concrete application to our MPC protocol. The final multiplication will be of a ciphertext with noise

$$\xi \cdot (B^2 + B_{\mathsf{Scale}}) + \frac{\xi \cdot B_{\mathsf{Ks}} \cdot q_1}{P},$$

and a ciphertext with noise B (namely $\mathfrak{c}_{\underline{1}}$). The input to the final key switch will have noise value approximately $\xi \cdot B^3$; we make this simplifying assumption which makes little difference to the final values. The output noise from the keyswitch is then equal to

$$\xi \cdot B^3 + B_{\mathsf{Scale}} + \frac{B_{\mathsf{Ks}} \cdot q_1}{P}.$$

We then perform a modulus switch to obtain a ciphertext as output of the multiplication gate with noise bounded by

$$\frac{\xi \cdot B^3 + B_{\mathsf{Scale}}}{p_1} + \frac{B_{\mathsf{Ks}} \cdot p_0}{P} + B_{\mathsf{Scale}}.$$

We again require this to be less than B, so we have now the cubic equation

$$\frac{\xi}{p_1} \cdot B^3 - B + \left(\frac{B_{\mathsf{Scale}}}{p_1} + \frac{B_{\mathsf{Ks}} \cdot p_0}{P} + B_{\mathsf{Scale}} \right) \leq 0.$$

Substituting in our existing estimate for P, namely $8 \cdot \xi \cdot B_{\mathsf{Ks}} \cdot q_{L-2} / B_{\mathsf{Scale}}$ we find the inequality is roughly equivalent to, assuming $L > 2$ and $p_1 \gg B_{\mathsf{Scale}}$ (i.e. $q_{L-2} \gg B_{\mathsf{Scale}} \cdot p_0$),

$$\frac{\xi}{p_1} \cdot B^3 - B + \frac{B_{\mathsf{Scale}}}{p_1} + B_{\mathsf{Scale}} \approx \frac{\xi}{p_1} \cdot B^3 - B + \left(\frac{B_{\mathsf{Scale}}}{p_1} + \frac{B_{\mathsf{Scale}} \cdot p_0}{8 \cdot q_{L-2}} + B_{\mathsf{Scale}} \right) \leq 0.$$

If we set $B \approx 2 \cdot B_{\mathsf{Scale}}$, then this means we have (approximately)

$$\frac{\xi}{p_1} \cdot 8 \cdot B_{\mathsf{Scale}}^3 - B_{\mathsf{Scale}} + \frac{B_{\mathsf{Scale}}}{p_1} \leq 0,$$

and so

$$p_1 \approx 8 \cdot (\xi + 1) \cdot B_{\mathsf{Scale}}^2 \tag{4}$$

will therefore guarantee the result.

We now need to estimate the size of p_0. Due to the above choice of p_1 the ciphertext to which we apply the distributed decryption has norm bound by B, to which we add on a random encryption of zero at level L. To do this we need to apply LowerLevel to this encryption of zero, and hence the noise level of the ciphertext we finally pass into $\mathsf{SHE.ShareDec}$ in our main MPC protocol has noise bounded by $B_{\mathsf{dec}} = 2 \cdot B$ This means that we require

$$q_0 = p_0 \geq 2^{\mathrm{sec}+2} \cdot B, \tag{5}$$

to ensure a valid distributed decryption.

Finally, set the Hamming weight h of the secret key sk to be 64 as in [31, 22]. Plugging this into our equations (1), (2), (3), (4), and (5), we obtain

$$\begin{aligned}
p_0 &\approx 309 \cdot 2^{\mathrm{sec}} \cdot p \cdot \sqrt{N}, \\
p_1 &\approx 107736 \cdot p^2 \cdot N, \\
p_i &\approx 1237 \cdot p \cdot \sqrt{N}, \text{ for } 2 \leq i \leq L-1, \\
p_L &\approx 2.34 \cdot \sigma \cdot \sqrt{N}, \\
P &\approx 0.404 \cdot 1237^L \cdot \sigma \cdot 2^{\mathsf{exp}} \cdot p^L \cdot N^{(L+3)/2}, \\
q_{L-1} &\approx 21.76 \cdot 1237^L \cdot 2^{\mathsf{exp}} \cdot p^{L+1} \cdot N^{(L+1)/2}.
\end{aligned}$$

The largest modulus used in our key switching matrices, i.e. the largest modulus used in an LWE instance, is given by $Q_{L-1} = P \cdot q_{L-1}$; where using the above estimates we have

$$Q_{L-1} \approx 8.79 \cdot 1237^{2 \cdot L} \cdot \sigma \cdot 4^{\mathsf{exp}} \cdot p^{2 \cdot L + 1} \cdot N^{L+2}.$$

Recall from Section 7.5 we have the following relationship between exp and our statistical security parameter sec; $\mathsf{exp} = \mathsf{sec} + \log_2(N)$. To ensure security we use the estimates of Lindner and Peikert [34], we require at the κ-bit security level we require

$$N > (\kappa + 110) \cdot \log(Q_{L-1}/\sigma)/7.2.$$

9 Estimating the Consumed Bandwidth

In Section 8 we determined the parameters for the instantiation of our SHE scheme using BGV by adapting the analysis from [22, 31]. In this section we use this parameter estimation to show that our MPC protocol can in fact give improved communication complexity compared to the standard MPC protocols, for relatively small values of the parameter L. We are interested in the communication cost of our online stage computation. To ease our exposition we will focus on the passively secure case from Section 4; the analysis for the active security case with $t < n/3$ is exactly the same (bar the additional cost of the exchange of zero-knowledge proofs for the input stage and the output stage). For the case of active security with $t < n/2$ we also need to add in the communication related to the dispute control strategy outlined in Section 6 for attaining robust SHE.ShareCombine with $t < n/2$; but this is a cost which is proportional to $\mathcal{O}(n^3)$.

To get a feel for the parameters from Section 8, we now specialise to the case of finite fields of size $p \approx 2^{64}$, statistical security parameter sec of 40, and for various values of the computational security level κ. Resolving the various inequalities (from Section 8), we then estimate in Table 1 the value of N, assuming a small value for n (we need to restrict to small n to ensure a large enough range in the PRF needed in the distributed decryption protocol; see Section 7.4).

L	$\kappa = 80$	$\kappa = 128$	$\kappa = 256$
2	16384	16384	32768
3	16384	16384	32768
4	16384	32768	32768
5	32768	32768	65536
6	32768	32768	65536
7	32768	32768	65536
8	32768	65536	65536
9	32768	65536	65536
10	65536	65536	65536

Table 1. The value of N for various values of κ and L

Since a Refresh gate requires the transmission of $n - 1$ elements (namely the decryption shares) in the ring R_{q_0} from party P_i to the other parties, the total communication in our protocol (in bits) is

$$|\mathbb{G}_R| \cdot n \cdot (n-1) \cdot |R_{q_0}|,$$

where $|R_{q_0}|$ is the number of bits needed to transmit an element in R_{q_0}, i.e. $N \cdot \log_2 p_0$. Assuming the circuit meets our requirement of being well formed, this implies that total communication cost for our protocol is

$$\frac{2 \cdot |\mathbb{G}_M| \cdot n \cdot (n-1) \cdot N \cdot \log_2 p_0}{L \cdot N} = \frac{2 \cdot n \cdot (n-1) \cdot |\mathbb{G}_M|}{L} \cdot \log_2(309 \cdot 2^{\mathrm{sec}} \cdot p \cdot \sqrt{N}).$$

Using the batch distributed decryption technique (of efficiently and parallely evaluating $t+1$ independent Refresh gates simultaneously) from Section 7.6 this can be reduced to

$$\mathsf{Cost} = \frac{4 \cdot n \cdot (n-1) \cdot |\mathbb{G}_M|}{L \cdot (t+1)} \cdot \log_2(309 \cdot 2^{\mathrm{sec}} \cdot p \cdot \sqrt{N}).$$

We are interested in the *overhead per multiplication gate*, in terms of equivalent numbers of finite field elements in $\mathbb{F}_p$, which is given by $\mathsf{Cost}/(|\mathbb{G}_M| \cdot \log_2 p)$, and the cost per party is $\mathsf{Cost}/(|\mathbb{G}_M| \cdot n \cdot \log_2 p)$.

At the 128 bit security level, with $p \approx 2^{64}$, and $\mathsf{sec} = 40$ (along with the above estimated values of N), this means for $n = 3$ parties, and at most $t = 1$ corruption, we obtain the following cost estimates:

L		2	3	4	5	6	7	8	9	10
Total Cost	$\mathsf{Cost}/(\lvert\mathbb{G}_M\rvert \cdot \log_2 p)$	12.49	8.33	6.31	5.05	4.21	3.61	3.19	2.84	2.55
Per party Cost	$\mathsf{Cost}/(\lvert\mathbb{G}_M\rvert \cdot n \cdot \log_2 p)$	4.16	2.77	2.10	1.68	1.40	1.20	1.06	0.94	0.85

Note for $L = 2$ our protocol becomes the one which requires interaction after every multiplication, for $L = 3$ we require interaction only after every two multiplications and so on. Note that most practical MPC protocols in the preprocessing model have a per gate per party communication cost of at least 2 finite field elements, e.g. [25]. Thus, even when $L = 5$, we obtain better communication efficiency in the online phase than traditional practical protocols in the preprocessing model with these parameters.

10 Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant EP/I03126X, and by Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079[5]. The second author was supported by an Trend Micro Ltd, and the fifth author was supported by in part by a Royal Society Wolfson Merit Award.

References

1. G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 483–501, 2012.
2. G. Asharov, A. Jain, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. *IACR Cryptology ePrint Archive*, 2011:613, 2011.
3. Z. BeerliováTrubíniová and M. Hirt. Efficient multi-party computation with dispute control. In *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 305–328, 2006.
4. Z. BeerliováTrubíniová and M. Hirt. Perfectly-secure MPC with linear communication complexity. In *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230, 2008.
5. E. Ben-Sasson, S. Fehr, and R. Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 663–680, 2012.
6. R. Bendlin and I. Damgård. Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems. In *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 201–218, 2010.
7. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188, 2011.
8. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206, 2008.
9. Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886, 2012.
10. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, pages 309–325. ACM, 2012.
11. Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS*, pages 97–106. IEEE, 2011.

[5] The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

12. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524, 2011.
13. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.
14. O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In *Financial Cryptography*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50, 2010.
15. R. Cleve. Limits on the security of coin flips when half the processors are faulty (Extended abstract). In *STOC*, pages 364–369. ACM, 1986.
16. T. M. Cover and J. A. Thomas. *Elements of Information theory*. Wiley, 2006.
17. R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362, 2005.
18. I. Damgård, M. Fitzi, E. Kiltz, J.B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304, 2006.
19. I. Damgård, Y. Ishai, M. Krøigaard, J.B. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 241–261, 2008.
20. I. Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 445–465, 2010.
21. I. Damgård, M. Keller, E. Larraia, C. Miles, and N.P. Smart. Implementing AES via an actively/covertly secure dishonest-majority mpc protocol. In *SCN*, volume 7485 of *Lecture Notes in Computer Science*, pages 241–263, 2012.
22. I. Damgard, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure mpc for dishonest majority – or: Breaking the SPDZ limits, 2013.
23. I. Damgård and J. B. Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264, 2003.
24. I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590, 2007.
25. I. Damgård, V. Pastro, N.P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, 2012.
26. I. Damgård and S. Zakarias. Constant-overhead secure computation for boolean circuits in the preprocessing model. In *TCC*, volume 7785 of *Lecture Notes in Computer Science*, pages 621–641, 2013.
27. M. Fitzi and M. Hirt. Optimally efficient multi-valued Byzantine agreement. In *PODC*, pages 163–168. ACM, 2006.
28. C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.
29. C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178. ACM, 2009.
30. C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482, 2012.
31. C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867, 2012.
32. O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.
33. M. Hirt and J.B. Nielsen. Robust multiparty computation with linear communication complexity. In *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 463–482, 2006.
34. R. Lindner and C. Peikert. Better key sizes (and attacks) for LWE-based encryption. In *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339, 2011.
35. A. López-Alt, E. Tromer, and V. Vaikuntanathan. Cloud-assisted multiparty computation from fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2011:663, 2011.
36. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, 2010.
37. J.B. Nielsen, P.S. Nordholt, C. Orlandi, and S.S. Burra. A new approach to practical active-secure two-party computation. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700, 2012.
38. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC*, pages 84–93, 2005.
39. N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *PKC*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443, 2010.
40. N.P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *To Appear in Designs, Codes and Cryptography*, 2012.

Actively Secure Private Function Evaluation

Payman Mohassel[1,2], Saeed Sadeghian[1], and Nigel P. Smart[3]

[1] Dept. Computer Science, University of Calgary,
pmohasse@ucalgary.ca, sadeghis@ucalgary.ca
[2] Yahoo Labs,
pmohassel@yahoo-inc.com
[3] Dept. Computer Science, University of Bristol,
nigel@cs.bris.ac.uk

Abstract. We propose the first general framework for designing actively secure private function evaluation (PFE), not based on universal circuits. Our framework is naturally divided into pre-processing and online stages and can be instantiated using any generic actively secure multiparty computation (MPC) protocol.

Our framework helps address the main open questions about efficiency of actively secure PFE. On the theoretical side, our framework yields the first actively secure PFE with linear complexity in the circuit size. On the practical side, we obtain the first actively secure PFE for arithmetic circuits with $O(g \cdot \log g)$ complexity where g is the circuit size. The best previous construction (of practical interest) is based on an arithmetic universal circuit and has complexity $O(g^5)$.

We also introduce the first linear Zero-Knowledge proof of correctness of "extended permutation" of ciphertexts (a generalization of ZK proof of correct shuffles) which maybe of independent interest.

Keywords. Secure Multi-Party Computation, Private Function Evaluation, Malicious Adversary, Zero-Knowledge Proof of Shuffle

1 Introduction

Private Function Evaluation (PFE) is a special case of Multi-Party Computation (MPC), where the parties compute a function which is a private input of one of the parties, say party P_1. The key additional security requirement is that all that should leak about the function to an adversary, who does not control P_1, is the size of the circuit (i.e. the number of gates and distinct wires within the circuit). Clearly, PFE follows immediately from MPC by designing an MPC functionality which implements a universal machine/circuit; thus the only open questions in PFE research are those of efficiency. Using universal circuits one can achieve complexity of $O(g^5)$ in case of arithmetic circuits [23] and $O(g \cdot \log g)$ for boolean circuits [26]. For ease of exposition we ignore the factors depending on the number of parties and the security parameters as they depend on the particular underlying MPC being used. We still provide some numbers for the specific SPDZ instantiation in section 5.

A number of previous work [1,2,4,12,14,15,16,17,22,24] have considered the design and implementation of more efficient general- and special-purpose private function evaluation. A major motivation behind these solutions (and PFE in general) is to hide the function being computed since it is proprietary, private or contains sensitive information. Some applications of interest considered in the literature are software diagnostic [4], medical applications [2], and intrusion detection systems [20].

But all prior solutions are in the semi-honest model and fail in the presence of an active adversary who does not follow the steps of the protocol (with the exception of the generic approach of applying an actively secure MPC to universal circuits). For example, a malicious party who does not own the function can cheat to learn the proprietary function or modify the outcome of computation without the function-holders' knowledge. Or a malicious function-holder, can learn information about honest parties' inputs.

This article the full version of an earlier article: Asiacrypt 2014, © IACR 2014, http://dx.doi.org/10.1007/978-3-662-45608-8_26.

One may question the need for actively secure PFE as the function-holder can cheat and use a malicious function, which reveals information about the other party's input. While we consider the general scenario in our protocols, there are common practical scenarios where the function-holder has no output in the computation, and therefore maliciously changing the function still does not let him learn anything even if he is actively cheating.

1.1 Our Contribution

In this work, we present the first general framework for designing actively secure PFE, not based on universal circuits. Our framework can be instantiated upon a generic actively secure MPC protocol satisfying quite general properties; namely that they are secret sharing based, actively secure (either robust or with aborts), can implement reactive functionalities, and have an ability to open various sharings securely, as well as generate (efficiently) sharings of random values. Suitable actively secure MPC protocols include BDOZ [3] and SPDZ [8] (for the case of arithmetic circuits and an arbitrary number of players with a dishonest majority), Tiny-OT [19] (for binary circuits and two players), or protocols such as that implemented in VIFF [7] utilizing Shamir secret sharing with a threshold of $t < n/3$.

Our framework helps address the main open questions about efficiency of actively secure PFE. On a theoretical note, we use it to show that actively secure PFE with linear complexity (in circuit size) is indeed feasible while avoiding strong primitives such as fully-homomorphic encryption (FHE).[4] On a practical note, we obtain a practical actively secure PFE for arithmetic circuit with $O(g \cdot \log g)$ complexity (a significant reduction from $O(g^5)$ [23]), and the first actively secure PFE in the information-theoretic setting.

Our Framework. Our framework can be seen as an extension of the new framework of [17] which is only secure against passive adversaries. The key idea in [17] is to divide the problem into two sub-problems, the problem of hiding the topology of the wiring between individual gates (topology hiding), and the problem of hiding exactly what gate is evaluated (gate hiding), i.e. an addition or a multiplication (or AND/OR/XOR in case of boolean circuits).

This framework yields better asymptotic and practical efficiency for passively secure PFE compared to the universal circuit approach (see [17] for a detailed efficiency comparison). An important open question is then how to extend their solution to the case of active adversaries efficiently. In this paper we do exactly that by providing a recipe for turning any actively secure MPC protocol that satisfies our general requirements into an actively secure PFE protocol.

Our framework operates in two phases, an offline phase and an online phase. As in the case of standard MPC in the pre-processing model, our offline phase is input independent but it depends on the function. The offline phase is use-once, in the sense that the data produced cannot be reused for multiple invocations of the online phase. We note that a similar function-dependent pre-processing model (referred to as *dedicated pre-processing*) was recently considered in [9]. Dedicated pre-processing is particularly natural in PFE applications where the sensitive/proprietary function stays fixed for a period of time and is used in multiple executions (clearly in the latter case we need to execute the pre-processing multiple times, but this can be done in advance). Of course, if one is not willing to count a function-dependent offline phase as valid, then our complexities would be the combination of the two phases. It maybe the case that our underlying MPC protocol is itself in the pre-processing model (e.g. [3,8,19]), in which case that pre-processing will be essentially independent of the input and function being evaluated. Our framework shows the feasibility of offline computation independent of inputs, which was not the case in [17]. We elaborate on the two phases next:

Offline Phase. Roughly speaking, our offline phase generates two vectors of random values, *maps* the second to a new vector using a mapping that captures the topology of the circuit (referred to as extended permutation

[4] Note that with the use of the right circuit-private FHE scheme [21], and appropriate ZK proofs for correctness of the computation on encrypted data, it is likely possible to achieve linear PFE based on FHE, but we are interested in the use of much weaker primitives such as singly homomorphic encryption.

in [17]), and subtracts the result from the first. The result of the subtraction (difference vector) is opened while the two original vectors are shared among the parties. The two random vectors are used as one-time pads of all the intermediate values in the circuit, while the "difference vector" is used by the function-holder to connect the output of one gate to the input of another without learning the values or revealing the circuit topology. The offline phase also generates one-time MACs of all the components of the "difference vector" computed above, using a fixed global MAC key. These MACs are used to check the function-holder's work in the online phase of the protocol. These steps commit P_1 privately to the topology of the circuit. We also privately commit P_1 to gate types, hence fully committing him to the function being computed.

Online Phase. Our online, or circuit evaluation, phase is very distinct from that deployed in the underlying MPC protocol we use. In existing instantiations of our underlying MPC protocol, parties evaluate gates on values whose secrecy is maintained due to the fact that one is working on secret shared values only. In our protocol the parties have public one-time pad encryptions of the values being computed on, but the encryption keys, which are the random values generated in the offline phase, remain secret-shared. Party P_1 (the function holder) then uses the random vectors computed in the offline phase to transform the encrypted output of one gate to the encrypted input of the upcoming gate while maintaining one-time MACs of all the values he computes. These MACs allow all other parties to check P_1's work without learning the circuit topology. These operations are carried out securely using the underlying MPC protocol.

In both the online and the offline phase, all parties check P_1's work by checking the MACs of the values he computes locally. If any of the MACs fail, in case of security with abort, parties can simply end the protocol. But in case of robust MPC (e.g. $t < n/3$ for robust information theoretically secure protocols) the protocol needs to continue without P_1. To achieve this, honest parties jointly recover P_1's function and play his role in the remainder of the protocol.

In our protocols, if any adversary deviates from the protocol then, except with negligible probability, the honest parties will either abort, or be able to recover from the introduced error. The exact response depends on the underlying MPC protocol on which our PFE protocol is built. In all cases the privacy of the honest players inputs is preserved, bar what can be obtained from the output of the private function chosen by player P_1. Note that P_1 may or may not be a recipient of output, but many application of PFE are concerned with scenarios where the function-holder has no output.

Efficient Instantiations. One can efficiently instantiate our online phase with a linear complexity, using any actively secure MPC satisfying our requirements. The main challenge, therefore, lies in efficient instantiation of the offline phase. It is possible to implement our offline phase using any actively secure MPC sub-protocol as well (by securely computing a circuit that performs the above mentioned task) but the resulting constructions would neither be linear nor constant-round.

- We introduce a instantiation with $O(g)$ complexity, proving the feasibility of linear actively secure PFE for the first time. Our main new technical ingredient is a linear zero-knowledge (ZK) proof of "correct extended permutation" of ElGamal ciphertexts. While linear ZK proofs of shuffles are well-studied, it is not clear how to extend the techniques to extended permutation (see our incomplete attempt in Appendix B) Instead, we propose a generic and linear solution that uses ZK proof of a correct shuffle in a black-box manner, and may be of independent interest. Our solution is based on the switching network construction of EP [17]. This construction consists of three components, two of which are permutation networks. Instead of evaluating switches, we use singly homomorphic encryption to evaluate each component, and then re-randomize. We use existing ZK proofs of shuffle to prove the correctness of first and third components which perform permutation. The middle component requires a separate compilation of ZK protocols. Note that generically applying ZK proofs to UC circuit evaluation does not provide a linear solution, and applying ZK proofs for the EP component also does not work. Our customized linear $\mathcal{ZK}_{\text{EP}}$ gets around these problems.
- We introduce a *constant-round* instantiation with $O(g \cdot \log g)$ complexity (contrast with $O(g^5)$ complexity for universal arithmetic circuits) that is also of practical interest. Our technique is itself an extension of ideas from [17]. In particular the basic algorithm is that of [17] for oblivious evaluation of a switching

network, but some care needs to be taken to make sure the protocol is actively secure. This is done by applying MACs to the data being computed on. However, instead of having the MAC values being secret shared (as in SPDZ) or kept secret (as in BDOZ and Tiny-OT), the MAC values are public with the keys remaining secret shared. Nevertheless, the MACs used are very similar to those used in the BDOZ and Tiny-OT protocols [3,19], since they are two-key MACs in which one key is a per message key and one is a global key. While using MAC's is quite standard for ensuring consistency of data, our efficient deployment in the framework is non-trivial and novel. For example, while addition of MACs in the offline phase is done using a generic MPC, the circuit evaluation (online phase) does not use an MPC. This is different from [17]'s approach and previous MPC work. General active security techniques can not be directly employed in this context. It is not clear how to use cut-and-choose in case of PFE, e.g. it is not clear how not to reveal the function in the opening, and there are additional components (i.e. EP) in a PFE protocol which cut-and-choose does not seem to resolve.

Efficiency Discussion. We emphasize that our linear complexity solution is a feasibility result at it was an open question whether active PFE with linear complexity in circuit size is possible given simple crypto primitive such as singly homomorphic encryption (as opposed FHE). Our "efficient" arithmetic PFE only requires $O(g \log g)$ multiplication gates and it is a significant improvement in comparison with applying of arithmetic MPC to universal arithmetic circuit of size $O(g^5)$ [23]. If we apply active secure MPC for arithmetic circuits to this universal circuit the complexity cannot get better than $O(g^5)$. One can turn an arithmetic circuit into a boolean circuit and use Valiant's boolean UC [26] to obtain a PFE. But this is highly inefficient, and therefore we do not discuss this in detail.

2 Notation and the Underlying MPC Protocol

We assume our function f to be evaluated will eventually be given by player P_1 as an arithmetic circuit over a finite field $\mathbf{F}_p$; note p may not necessarily be prime. We let $\mathbf{g}(f)$ denote the number of gates in the circuit representing f. For gates with fan-out greater than one, we count each seperate output wire as a different wire. We also select a value k such that $p^k > 2^{\mathsf{sec}}$, where sec is the security parameter; this is to ensure security of our MAC checking procedure in the online phase.

We assume n parties $P_1, \ldots, P_n$, of which an adversary may corrupt (statically) up to t of them; the value of t being dependent on the specific underlying MPC protocol. The corrupted adversaries could include party P_1. The MPC protocol should implement the functionality described in Figure 1. This functionality is slightly different from standard MPC functionalities in that we try to capture both the honest majority and the dishonest majority setting; and in the latter setting the adversary can force the functionality to abort at any stage of the computation and not just the output. We also introduce another operation called **Cheat** which will be useful in what follows.

It is clear that modern actively secure MPC protocols such as [7,8,19], implement this functionality in different settings. Thus various different settings (i.e. different values of n, p and t) will be able to be dealt with in our resulting PFE protocol by simply plugging in a different underlying MPC protocol. To ease exposition later we express our MPC protocol as evaluating functions in the finite field $\mathbf{F}_{p^k}$. Clearly such an MPC protocol can be built out of one which evaluates functions over the base finite field $\mathbf{F}_p$.

To ease notation in what follows we shall let $[varid]$ denote the value stored by the functionality under $(varid, a)$; and will write $[z] = [x] + [y]$ as a shorthand for calling **Add** and $[z] = [x] \cdot [y]$ as a shorthand for calling **Multiply**. And by abuse of notation we will let $varid$ denote the value, x, of the data item held in location $(varid, x)$.

3 Our Active PFE Framework

In this section we describe our active PFE framework in detail. We start by describing the offline functionality which pre-processes the function/circuit the parties want to compute (Section 3.1). Then, in Section 3.2,

Functionality $\mathcal{F}_{\text{MPC}}$

The functionality consists of seven externally exposed commands **Initialize**, **Cheat**, **Input Data**, **Random**, **Add**, **Multiply**, and **Output** and one internal subroutine **Wait**.

Initialize: On input $(init, p, k, flag)$ from all parties, the functionality activates and stores p and k; and a representation of $\mathbf{F}_{p^k}$. The value of *flag* is assigned to the variable dhm, to signal whether the MPC functionality should operate in the dishonest majority setting. The set of "valid" players is initially set to all players. In what follows we denote the set of adversarial players by $\mathcal{A}$.

Cheat: This is a command which takes as input a player index i, it models the case of (most) robust MPC protocols in the honest majority case. On execution the functionality aborts if dhm is set to *true*. Otherwise the functionality waits for input from all players. If a majority of the players return *OK* then the functionality reveals all inputs made by player i, and player i is removed from the list of "valid" players (the functionality continues as if player i does not exist).

Wait: This does two things depending on the value of dhm.

- If dhm is set to *true* then it waits on the environment to return a *GO*/*NO-GO* decision. If the environment returns *NO-GO* then the functionality aborts.
- If dhm is set to *false* then it waits on the environment. The environment will either return *GO*, in which case it does nothing, or the environment returns a value $i \in \mathcal{A}$, in which case Cheat(i) is called.

Input Data: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with *varid* a fresh identifier, the functionality stores $(varid, x)$. The functionality then calls **Wait**.

Random: On command $(random, varid)$ from all parties, with *varid* a fresh identifier, the functionality selects a random value r in $\mathbf{F}_{p^k}$ and stores $(varid, r)$. The functionality then calls **Wait**.

Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x + y)$. The functionality then calls **Wait**.

Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x \cdot y)$. The functionality then calls **Wait**.

Output: On input $(output, varid)$ from all honest parties (if *varid* is present in memory), the functionality retrieves $(varid, x)$ and outputs it to the environment. The functionality then calls **Wait**, and only if **Wait** does not abort then it outputs x to all players.

Fig. 1: The required ideal functionality for MPC

we show that given a secure implementation of $\mathcal{F}_{\text{OFFLINE}}$, one can efficiently (linear complexity) construct an actively secure PFE based on any actively secure MPC. We postpone efficient instantiations of $\mathcal{F}_{\text{OFFLINE}}$ to later sections.

3.1 The Function Pre-Processing (Offline) Phase

In this section we detail the requirements of our pre-processing step once player P_1 has decided on the function f to be evaluated. P_1 is only required to enter a valid circuit, equivalent to his function f into the protocol. Each non-output wire w in the circuit is connected at one end (which we shall call the *outgoing wire or left point*) to a source, this is either the output of a (non-output) gate or an input wire. Conversely each non-output wire is connected at the other end (which we shall call the *incoming wire or right point*) to a destination point which is always an input to a gate. We denote the number of distinct Incoming Wires on the right by $\mathsf{iw}(f)$. We let $\mathsf{ow}(f)$ denote the number of Outgoing Wires on the left. Note that $\mathsf{iw}(f) = 2g$ and $\mathsf{ow}(f) = n + g - o$ where o is the number of output gates in the circuit. Since we are dealing with arbitrary fan out we have that $\mathsf{ow}(f) \leq \mathsf{iw}(f)$.

Functionality $\mathcal{F}_{\text{OFFLINE}}$

Initialize: As for $\mathcal{F}_{\text{MPC}}$.
Wait: As for $\mathcal{F}_{\text{MPC}}$.
Input Data: As for $\mathcal{F}_{\text{MPC}}$.
Cheat: As for $\mathcal{F}_{\text{MPC}}$.
Random: As for $\mathcal{F}_{\text{MPC}}$.
Add: As for $\mathcal{F}_{\text{MPC}}$.
Multiply: As for $\mathcal{F}_{\text{MPC}}$.
Output: As for $\mathcal{F}_{\text{MPC}}$.
Input Function: On input $(\mathit{inputfunction}, \pi, f)$ from player P_1 the functionality performs the following operations

- The functionality calls (random, K).
- If f is not a valid arithmetic circuit then the functionality aborts.
- For $i \in \{1, \ldots, \mathsf{iw}(f)\}$ the functionality calls (random, r_i) and (random, s_i).
- For $j \in \{1, \ldots, \mathsf{ow}(f)\}$ the functionality calls (random, l_j) and (random, t_j).
- The functionality then computes, for all $i \in \{1, \ldots, \mathsf{iw}(f)\}$

$$[p_i] = [r_i] - [\ell_{\pi(i)}], \quad [q_i] = ([s_i] - [t_{\pi(i)}]) + ([r_i] - [\ell_{\pi(i)}]) \cdot [K]$$

- The functionality then outputs (p_i, q_i) to all players, for $i \in \{1, \ldots, \mathsf{iw}(f)\}$, by calling (output, p_i) and (output, q_i).
- For $i \in \{1, \ldots, g\}$ the functionality calls $(\mathit{input}, P_1, G_i, 0)$ if gate i in the description of f is an addition gate, and $(\mathit{input}, P_1, G_i, 1)$ if gate i is a multiplication gate.

Fig. 2: The required ideal functionality for the Offline Phase

To fully capture the topology of the circuit we give each outgoing wire and incoming wire in the circuit a unique label. The labels for the outgoing wires will be $\{1, \ldots, \mathsf{ow}(f)\}$ starting from the input wires and then moving to the output wires of each gate in a topological order decided by P_1, whilst the labels for the incoming wires will be $\{1, \ldots, \mathsf{iw}(f)\}$ labelling the input wires to each gate in the same topological order. The topology is then defined by a mapping from outgoing wires to incoming wires and is called an "extended permutation" in [17]as demonstrated in Figure 3. We denote the inverse of this mapping by a function π from $\{1, \ldots, \mathsf{iw}(f)\}$ onto $\{1, \ldots, \mathsf{ow}(f)\}$. If w is a wire in the circuit with incoming wire label i, then it's outgoing wire label is given by $j = \pi(i)$.

To execute the function pre-processing, player P_1 on input of f determines a mapping π corresponding to f. The offline phase functionality $\mathcal{F}_{\text{OFFLINE}}$ which is described in Figure 2, extends the $\mathcal{F}_{\text{MPC}}$ functionality

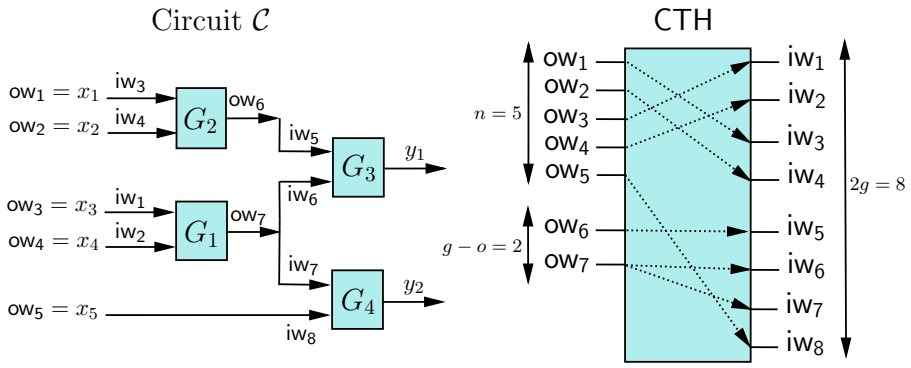

Fig. 3: An example circuit and the corresponding mapping [17]

of Figure 1 by adding an additional operation **Input Function**. The **Input Function** generates a vector of random (but correlated) values and their one-time MACs using a fixed global MAC key K. In particular, the functionality first stores a vector of random values (r_i) for each incoming wire and another vector of random values (ℓ_i) for the outgoing wires in the circuit. These random values will play the role of "pads" for one-time encryption of the computed wire values in the online phase. The functionality then computes p_i, the difference between each outgoing wire's value r_i and the corresponding incoming wires' value $\ell_{\pi(i)}$, and reveals p_i to all parties. This difference vector will allow P_1 to maintain one-time encryption of each wire value in the online phase without revealing the circuit topology. Additional random values (s_i, t_i) and the global MAC key K are used to compute one-time MACs of each p_i, namely q_i. These MACs will be used to check P_1's actions in the online phase. The **Input Function** also commits P_1 to the function of each gate in his circuit by storing a bit (0 for addition and 1 for multiplication) for each gate.

3.2 The Function Evaluation (Online) Phase

We can now present our framework for actively secure PFE. We wish to implement the functionality in Figure 4. We express the functionality as evaluating a function f provided by P_1 which takes as input n inputs in $\mathbf{F}_{p^k}$, one from each player. Again we present the functionality in both the honest majority and the dishonest majority settings.

Realizing $\mathcal{F}_{\text{Online}}$ Given $\mathcal{F}_{\text{Offline}}$ and $\mathcal{F}_{\text{MPC}}$ A generic instantiation of $\mathcal{F}_{\text{OFFLINE}}$ based on any MPC is give in Figure 6. The idea is to work with *one-time pad* encryptions of the values for all intermediate wires and the corresponding one-time MACs. Here, the pads $(r, \ell, s, t$ values), as well as the MAC Key K are generated by the offline functionality, and shared among the parties so no party can learn intermediate values or forge MACs on his own.

In more detail, the protocol proceeds as follows. Initially, parties compute one-time encryption of the input values to the circuit (pads are the corresponding ℓ values). Then, the following process is repeated for every gate in the circuit until every gate is processed. Parties then open the outcome of the output gates as their final result.

For each gate, party P_1 uses the "difference vectors" (p_i values) from the offline phase to transform the one-time encryption of output of the previous gate to the one-time encryption of input of the current gate (the result is denoted by d_{i_0}, d_{i_1} for the i-th gate.), without revealing the topology or learning the actual wire values. This is diagrammatically presented in Figure 5 to aid the reader. A similar transformation is done on MACs of the wire values (using q_i values) in order to keep P_1 honest in his computation (denoted by m_{i_0}, m_{i_1}).

Then, the protocol proceeds by jointly removing the one-time pads for the two inputs of the current gate and evaluating it together in order to compute a shared output z_i. Note that in this gate evaluation the gate

Functionality $\mathcal{F}_{\text{ONLINE}}$

Initialize: On input $(init, p, k, flag)$ from all players, the functionality activates and stores p and k; and a representation of $\mathbf{F}_{p^k}$. The value of *flag* is assigned to the variable dhm, to signal whether the underlying MPC functionality should operate in the dishonest majority setting.

Wait: If dhm is set to *false* then this does nothing. Otherwise it waits on the environment to return a *GO/NO-GO* decision. If the environment returns *NO-GO* then the functionality aborts.

Input Function: On input $(inputfunction, f)$ from player P_1 the functionality stores $(function, f)$. The functionality now calls **Wait**.

Input Data: On input $(input, P_i, x_i)$ from player P_i the functionality stores $(input, i, x_i)$. The functionality now calls **Wait**.

Output: On input $(output)$ from all honest players the functionality retrieves the data x_i stored in $(input, i, x_i)$ for $i \in \{1, \ldots, n\}$ (if all do not exist then the functionality aborts). The functionality then retrieves f from $(function, f)$ and computes $y = f(x_1, \ldots, x_n)$ and outputs it to the environment (or aborts if $(function, f)$ has not been stored). The functionality now calls **Wait**. Only on a successful return from **Wait** will the functionality output y to all players.

Fig. 4: The required ideal functionality for PFE

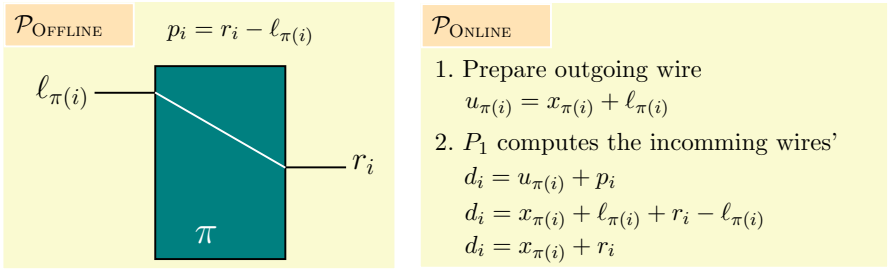

Fig. 5: Transformation of one-time encryption of an outgoing wire to the one-time encryption of an incoming wire using the values computes in $\mathcal{P}_{\text{OFFLINE}}$ protocol.

type G_i is secret and shared among the players. This step can be performed using the $\mathcal{F}_{\text{MPC}}$ operations. Then, parties compute a one-time encryption of z_i using the corresponding ℓ value as the pad, and denote the result by u_j, just a relabeling where j is the outgoing wire's label of the output wire of the gate (note that $j = n + i$ since the outgoing wires are labeled starting with the n input wires and then the output wire of each gate).

Note, that if P_1 tries to deviate from the protocol in his local computation (i.e. when he connects outgoing wires to incoming wires) the generated MACs will not pass the jointly performed verifications and he will be caught. In that case, either the protocol aborts (in the case of dishonest majority) or his input (i.e. the function) is revealed (in the case of honest majority).

This leads to the following theorem, whose proof is given in Appendix F.

Theorem 1. *In the $\mathcal{F}_{\text{OFFLINE}}$-hybrid model the protocol in Figure 6 securely implements the PFE functionality in Figure 4, with complexity $O(g)$.*

4 Implementing $\mathcal{F}_{\text{Offline}}$ with Linear Complexity

In this section we give a linear instantiation of the offline phase of the framework. Since our online phase has linear complexity, a linear offline phase implementation leads to a linear actively secure PFE. The main challenge in obtaining a linear solution is to design a linear method for applying the extended permutation

Protocol $\mathcal{P}_{\text{ONLINE}}$

The protocol is described in the $\mathcal{F}_{\text{OFFLINE}}$-hybrid model.

Input Function: Player P_1 given f selects the switching network mapping π and then calls $(\mathit{inputfunction}, \pi, f)$ on the functionality $\mathcal{F}_{\text{OFFLINE}}$.

Input Data: On input $(\mathit{input}, P_i, x_i)$ from player P_i the protocol executes the (input, i, x_i) operation of the functionality $\mathcal{F}_{\text{OFFLINE}}$.

Output: The evaluation of the function proceeds as follows; where for ease of exposition we set $x_{\pi(h)} = y_h$ for all h, i.e. if a wire has input x_i on the left (as outgoing wire) then it has the same value y_h on the right (as incoming wire) where $i = \pi(h)$

– **Preparing Inputs to the Circuit:**

- For each input wire i ($1 \le i \le n$) the players execute $[u_i] = [x_i] + [\ell_i]$, where i is the outgoing wire's label corresponding to that input wire, and $[v_i] = [t_i] + ([x_i] + [\ell_i]) \cdot [K]$ using the $\mathcal{F}_{\text{MPC}}$ functionality available via $\mathcal{F}_{\text{OFFLINE}}$.
- Parties then call (output, u_i) and (output, v_i) to open $[u_i]$ and $[v_i]$.

– **Evaluating the Circuit:** For every gate $1 \le i \le g$ in the circuit players execute the following (here we assume that the gates are indexed in the same topological order P_1 chose to determine π):

- P_1 **Prepares the Two Inputs for Gate** i**.**
  - Note that the two input wires for gate i have incoming wire labels $i_0 = 2i - 1$ and $i_1 = 2i$, and the (u, v) value for their corresponding outgoing wire labels are already determined, i.e. $u_{\pi(i_j)}$ and $v_{\pi(i_j)}$ are already opened for $j \in \{0, 1\}$.
  - Player P_1 computes, for $j = 0, 1$,
$$\begin{aligned} d_{i_j} = u_{\pi(i_j)} + p_{i_j} &\doteq (y_{i_j} + \ell_{\pi(i_j)}) + (r_{i_j} - \ell_{\pi(i_j)}) \\ &\doteq y_{i_j} + r_{i_j}, \\ m_{i_j} = v_{\pi(i_j)} + q_{i_j} &\doteq (t_{\pi(i_j)} + (y_{i_j} + \ell_{\pi(i_j)}) \cdot K) \\ &\quad + \big((s_{i_j} - t_{\pi(i_j)}) + (r_{i_j} - \ell_{\pi(i_j)})) \cdot K\big) \\ &\doteq s_{i_j} + (y_{i_j} + r_{i_j}) \cdot K. \end{aligned}$$
  - Player P_1 then broadcasts the values d_{i_j} and m_{i_j} to all players.
- **Players Check** P_1**'s Input Preparation.**
  - All players then use the $\mathcal{F}_{\text{MPC}}$ operations available (via the interface to the $\mathcal{F}_{\text{OFFLINE}}$ functionality) so as to store in the $\mathcal{F}_{\text{MPC}}$ functionality the values $[n_{i_j}] = [s_{i_j}] + (y_{i_j} + r_{i_j}) \cdot [K]$. The value is then opened to all players by calling $(\mathit{Output}, n_{i_j})$.
  - If $n_{i_j} \neq m_{i_j}$ then the players call **Cheat**(1) on the $\mathcal{F}_{\text{MPC}}$ functionality. This will either abort, or return the input of P_1 (and hence the function), in the latter case the players can now proceed with evaluating the function using standard MPC and without the need for P_1 to be involved.
- **Players Jointly Evaluate Gate** i**.**
  - The players store the value $[y_{i_j}] = d_{i_j} - [r_{i_j}]$ in the $\mathcal{F}_{\text{MPC}}$ functionality.
  - The $\mathcal{F}_{\text{MPC}}$ functionality is then executed so as to compute the output of the gate as
$$[z_i] = (1 - [G_i]) \cdot ([y_{i_0}] + [y_{i_1}]) + [G_i] \cdot [y_{i_0}] \cdot [y_{i_1}].$$
  - Note that the outgoing wire label corresponding to the output wire of the ith gate is $j = n + i$ so we just relabel $[z_i]$ to $[z_j]$.
  - If G_i is an output gate, players call (Output, z_i) to obtain z_i, disregard next steps and continue to evaluate next gate.
  - The players compute via the MPC functionality $[u_j] = [z_j] + [\ell_j]$.
  - The players call (Output, u_j) so as to obtain u_j.
  - The players then compute via the MPC functionality
$$[v_j] = [t_j] + u_j \cdot [K] \doteq [t_j + (z_j + \ell_j) \cdot K].$$
  - The players call (Output, v_j) so as to obtain v_j.

Fig. 6: The Protocol for implementing PFE

π to values $\{[\ell_i]\}$ and $\{[t_i]\}$ to produce shared values $\{[\ell_{\pi(i)}]\}$ and $\{[t_{\pi(i)}]\}$. In the semi-honest case [17], linear complexity solution for this problem is achieved by employing a singly homomorphic encryption. The shared values are jointly encrypted; P_1 applies the extended permutation to the resulting ciphertexts and re-randomizes them in order to hide π; parties jointly decrypt in order to obtain the shares of the resulting plaintexts. To obtain active security, we need to make each step of the following computation actively secure:

1. Players encrypt the shared input (all of which lie in $\mathbf{F}_{p^k}$) using an encryption scheme, with respect to a public key for which the players can execute a distributed decryption protocol. The resulting ciphertexts are sent to P_1.
2. Player P_1 applies the EP and re-randomizes the ciphertexts and sends them back. He then uses the $\mathcal{ZK}_{\text{EP}}$ protocol to prove his operation has been done correctly.
3. The players then decrypt the permuted ciphertexts and recover shares of the plaintexts.

To implement the first and last steps we use an an instantiation based on ElGamal encryption, see Appendix A. The middle step is more tricky, and we devote the rest of this section to describing this. For the middle step we need a linear zero-knowledge protocol to prove that P_1 applied a valid EP to the ciphertexts. Proof of a correct shuffle is a well studied problem in the context of Mix-Nets, and linear solutions for it exist [11]. As discussed in Appendix B , however, extending these linear proofs to the case of extended permutations faces some subtle difficulties which we leave as an open question. Instead we aim for a more general construction that uses the currently available proofs of shuffling, in a black-box way.

4.1 Linear $\mathcal{ZK}_{\text{EP}}$ Protocol

After players compute the encryption of the shared inputs, P_1 knowing the circuit topology, applies the corresponding extended permutation to the ciphertexts. He then re-randomizes the ciphertexts and then "opens" the ciphertexts. Next, we give a linear zero-knowledge protocol $\mathcal{ZK}_{\text{EP}}$, which enables P_1 to prove the correctness of his operation (i.e final ciphertexts are the result of P_1 applying a valid EP to the input ciphertexts). As our first attempt we considered the possibility of extending existing linear proofs of shuffle to get linear proofs of extended permutation. While plausible there are subtle difficulties that need to be addressed. For more details regarding our attempt on extending the method of Furukawa [11,10], refer to Appendix B . We leave this approach as an open problem. Instead we give a more general construction which makes black-box calls to proof of shuffle. This construction is inspired by the switching network construction of EP given in [17]. We first revisit the extended permutation construction of [17].

Assume the EP mapping represented by the function: $\pi : \{1...n\} \rightarrow \{1...m\}$ (Which maps m input wires to n output wires ($n \geq m$)). Note that in this section we use n and m to denote the size of EP. In a switching network, the number of inputs and outputs are the same, therefore, the construction takes m real inputs of the EP and $n - m$ additional *dummy inputs*. The construction is divided into three components. Each component takes the output of the previous one as input. Instead of applying the EP in one step, P_1 applies each component separately and uses a zero-knowledge protocol to prove its correctness. Figure 7 demonstrates the components. Next, we describe each component and identify the required ZK proof.

Table 1 lists the zero-knowledge protocols that we make a black-box use in our $\mathcal{ZK}_{\text{EP}}$ protocol. Note that we use P and Q for our EC instantiation instead of g and h.

- **Dummy-value placement component:** This takes the real and dummy ciphertexts as input and for each ciphertexts of a real value that is mapped to k different outputs according to π, outputs the real ciphertexts followed by $k-1$ dummy ciphertexts. This is repeated for each real ciphertext. The resulting output ciphertexts are all re-randomized. The dummy replacement step can be seen as a shuffling of the input ciphertexts. We use a proof of correct shuffle, $\mathcal{ZK}_{\text{Shuffle}}$, for correctness of this component.
- **Replication component:** This takes the output of the previous component as input. It directly outputs each real ciphertext but replaces each dummy ciphertext with an encryption of the real input that precedes it. At the end of this step, we have the necessary copies for each real input and the dummy inputs are eliminated. Naturally, all the ciphertexts are re-randomized. To prove correctness of this step,

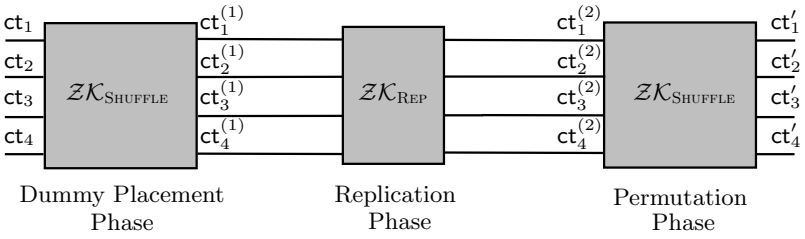

Fig. 7: EP construction. Components' names are written underneath. The zero-knowledge protocol for each component is written inside it's component box.

$\mathcal{ZK}$ Protocol	Relation/Language	Ref.
$\mathcal{ZK}_{\text{SHUFFLE}}(\{\mathfrak{ct}_i\},\{\mathfrak{ct}'_i\})$	$\mathcal{R}_{\text{SHUFFLE}} = \{(G,g,h,\{\mathfrak{ct}_i\},\{\mathfrak{ct}'_i\})\mid \exists\pi, \text{st.}$ $C_1'^{(i)} = g^{r_i} {C_1}^{(\pi(i))} \wedge C_2'^{(i)} = h^{r_i} {C_2}^{(\pi(i))} \wedge \pi \text{ is perm.}\}$	[11]
$\mathcal{ZK}_{\text{EQ}}(\mathfrak{ct}_1,\mathfrak{ct}_2)$	$\mathcal{R}_{\text{EQ}} = \{(G,g,h,\mathfrak{ct}_i = \langle\alpha_i,\beta_i\rangle_{i\in\{1,2\}})\mid \exists(m_1,m_2), \text{st.}$ $\alpha_i = g^{r_i} \wedge \beta_i = m_i h^{r_i} \wedge m_1 = m_2\}$	[5]
$\mathcal{ZK}_{\text{NO}}(\mathfrak{ct})$	$\mathcal{L}_{\text{NO}} = \{(G,g,h,\mathfrak{ct} = \langle\alpha,\beta\rangle)\mid \exists(m_1 \neq 1), \text{st.}$ $\alpha = g^r \wedge \beta = m_1 h^r\}$	[13]

Table 1: List of zero-knowledge protocols used in our $\mathcal{ZK}_{\text{EP}}$ protocol. Generator g and public key $h = g^{sk}$.

we need ZK proofs that the i-th output ciphertext has a plaintext equal to that of either the i-th input ciphertext or $(i-1)$-th output ciphertext (these can be achieved using protocol $\mathcal{ZK}_{\text{EQ}}$ defined in Table 1 as a building block). But this is not sufficient to guarantee a correct EP, as we also have to make sure that after the replication component there are no dummy ciphertexts left. For this, we assume that all dummy ciphertexts are encryptions of one. Then for each output ciphertext in the replication component we use a protocol $\mathcal{ZK}_{\text{NO}}$, i.e. a ZK proof that the underlying plaintext is not one. The $\mathcal{ZK}_{\text{REP}}$ zero-knowledge protocol, is a compilation of three ZK protocols, two checking for equality of ciphertexts and one checking the inequality of plaintext to one.

– **Permutation component:** This takes the output of the replication component as input and permutes each element to its final location as prescribed by π. We again use the proof of correct shuffle, $\mathcal{ZK}_{\text{SHUFFLE}}$. for this component.

$\mathcal{ZK}_{\text{EP}}$ Protocol description We assumed the inputs to the $\mathcal{ZK}_{\text{EP}}$, to be the outputs of our encryption functionality. Prover applies the extended permutation to the ciphertexts $(\mathfrak{ct}_1,\ldots,\mathfrak{ct}_n)$, where $\mathfrak{ct}_i = (C_1^{(i)}, C_2^{(i)})$. The prover obtains a re-randomized $(\mathfrak{ct}'_1,\ldots,\mathfrak{ct}'_n)$, where $\mathfrak{ct}'_i = (C_1'^{(i)}, C_2'^{(i)})$. We employ the techniques of Cramer et al. [6], to combine HVZK proof systems corresponding to each component, at no extra cost, into HVZK proof systems of the same class for any (monotonic) disjunctive and/or conjunctive formula over statements proved in the component proof systems. Figure 8 shows the complete description of our $\mathcal{ZK}_{\text{EP}}$ protocol. Note that we can choose dummy values from any set of random values S_d and substitute the $\mathcal{ZK}_{\text{NO}}(x)$ with $\vee_{\forall y\in S_d}(\mathcal{ZK}_{\text{EQ}}(x,y))$.

Theorem 2. *The protocol described in Figure 8 is HVZK proof of an extended permutation π, $(\mathfrak{ct}_1,\ldots,\mathfrak{ct}_n)$ and $(\mathfrak{ct}'_1,\ldots,\mathfrak{ct}'_n)$ in the $\mathcal{ZK}_{\text{SHUFFLE}}$, $\mathcal{ZK}_{\text{EQ}}$, $\mathcal{ZK}_{\text{NO}}$ hybrid model, for the following relation:*

$$\mathcal{R}_{\text{EP}} = \{(G,g,h,\{\mathfrak{ct}_i\},\{\mathfrak{ct}'_i\})\mid \exists\pi, \textit{st. } C_1'^{(i)} = g^{r_i} {C_1}^{(\pi(i))} \wedge C_2'^{(i)} = h^{r_i} {C_2}^{(\pi(i))} \wedge \pi \textit{ is EP.}\}$$

Proof. Following is a proof sketch. We show if the construction of EP from [17] is correct, then the $\mathcal{ZK}_{\text{EP}}$ protocol is a HVZK proof for EP. The goal of first two components is to prepare enough copies of each element. This implies that after the second component no dummy elements should be remained and no new elements are introduced. $\mathcal{ZK}_{\text{SHUFFLE}}$ and $\mathcal{ZK}_{\text{REP}}$ guarantee these two. $\mathcal{ZK}_{\text{SHUFFLE}}$ makes sure no additional

Protocol $\mathcal{ZK}_{\text{EP}}(\{\mathfrak{ct}_i\}, \{\mathfrak{ct}'_i\})$

Shared Input: Ciphertexts $(\mathfrak{ct}_1, \ldots, \mathfrak{ct}_n)$
P_1**'s Input:** Extended permutation π
P_1 **Evaluates the components.**

- Player P_1 finds the corresponding permutation π_1, and π_2 for Dummy-placement component and permutation components.
- P_1 applies the Dummy-placement component to $(\mathfrak{ct}_1, \ldots, \mathfrak{ct}_n)$, and re-randomizes to find $(\mathfrak{ct}_1^{(1)}, \ldots, \mathfrak{ct}_n^{(1)})$.
- P_1 applies the Replication component to $(\mathfrak{ct}_1^{(1)}, \ldots, \mathfrak{ct}_n^{(1)})$, and re-randomizes them to find $(\mathfrak{ct}_1^{(2)}, \ldots, \mathfrak{ct}_n^{(2)})$.
- P_1 applies the permutation component to $(\mathfrak{ct}_1^{(2)}, \ldots, \mathfrak{ct}_n^{(2)})$, and re-randomizes them to find $(\mathfrak{ct}'_1, \ldots, \mathfrak{ct}'_n)$.

P_1 **Computes the ZK proofs and sends everything**

- Player P_1 uses the $\mathcal{ZK}_{\text{SHUFFLE}}(\{\mathfrak{ct}_i\}, \{\mathfrak{ct}_i^{(1)}\})$ and $\mathcal{ZK}_{\text{SHUFFLE}}(\{\mathfrak{ct}_i^{(2)}\}, \{\mathfrak{ct}'_i\})$ protocols to produce proof of correctness for his evaluation of Dummy-placement component and permutation component.
- Player P_1 used the $\mathcal{ZK}_{\text{REP}}(\{\mathfrak{ct}_i^{(1)}\}, \{\mathfrak{ct}_i^{(2)}\})$ to produce proof of correctness for his evaluation of Replication component as follows(using [6] for combination) (and $\mathcal{ZK}^1_{\text{REP}} = \mathcal{ZK}_{\text{NO}}\left(\mathfrak{ct}_1^{(2)}\right) \wedge \mathcal{ZK}_{\text{EQ}}(\mathfrak{ct}_1^{(1)}, \mathfrak{ct}_1^{(2)})$):
  - For $2 \leq i \leq n$:

$$\mathcal{ZK}^i_{\text{REP}} = \left(\mathcal{ZK}_{\text{EQ}}(\mathfrak{ct}_i^{(1)}, \mathfrak{ct}_i^{(2)}) \vee \mathcal{ZK}_{\text{EQ}}(\mathfrak{ct}_{i-1}^{(2)}, \mathfrak{ct}_i^{(2)})\right) \wedge \mathcal{ZK}_{\text{NO}}\left(\mathfrak{ct}_i^{(2)}\right)$$

    .
  - $\mathcal{ZK}_{\text{REP}} = \wedge_{i=1,\ldots,n}(\mathcal{ZK}^i_{\text{REP}})$
- Player P_1 sends $(\mathfrak{ct}_1^{(1)}, \ldots, \mathfrak{ct}_n^{(1)})$, $(\mathfrak{ct}_1^{(2)}, \ldots, \mathfrak{ct}_n^{(2)})$, $(\mathfrak{ct}'_1, \ldots, \mathfrak{ct}'_n)$ and all proofs to other players.

Players verify P_1 **operations**

- Players verify P_1's operations by verifying the the proofs sent by P_1.

Fig. 8: The protocol for zero-knowledge proof of extended permutation.

elements are introduced in the first component. $\mathcal{ZK}_{\text{REP}}$ ensures each element is one of the input pairs to the second component. This makes sure no new elements are introduced in this step. Furthermore, it checks using $\mathcal{ZK}_{\text{NO}}$ for remaining dummy elements. Note that the EP construction does not require dummy-placement phase to necessarily arrange the elements in any order, and as long as we have satisfied the two mentioned properties, application of any permutation component, results in a valid EP, and also a valid circuit topology. $\mathcal{ZK}_{\text{SHUFFLE}}$ is used to check the final component. This sums up the proof. Finally we employ the techniques of Cramer et al. [6], to combine HVZK proof systems corresponding to each component, at no extra cost, into HVZK proof systems of the same class. Note that we make a black-box call to underlying ZK proof systems.

Offline Protocol Having all the parts of the puzzle, we can give the complete $O(g)$ protocol for the offline phase. Figure 9 shows the description, with the proof of security given in Appendix C.

5 A practical Implementation of $\mathcal{F}_{\text{Offline}}$ with $O(g \cdot \log g)$ Complexity

A $O(g \cdot \log g)$ protocol to implement $\mathcal{F}_{\text{OFFLINE}}$ is given in Figure 13 and Figure 14 (see Appendix D), and is in the $\mathcal{F}_{\text{MPC}}$-hybrid model. Following the ideas in [17], we implement the functionality via secure evaluation of a *switching network* corresponding to the mapping π_f.

Switching Networks. A switching network SN is a set of interconnected switches that takes N inputs and a set of selection bits, and outputs N values. Each *switch* in the network accepts two ℓ-bit strings as input and outputs two ℓ-bit strings. In this paper we need to use a switching network that contains two switch types. In the first type (*type 1*), if the selection bit is 0 the two inputs remain intact and are directly fed to the two outputs, but if the selection bit is 1, the two input values swap places. In the second type (*type 2*), if the selection bit is 0, as before, the inputs are directly fed to outputs but if it is 1, the value of the first input is used for both outputs. For ease of exposition, in our protocol description we assume that all switches are of type 1, but the protocol can be easily extended to work with both switch types.

Linear Implementation of Protocol $\mathcal{P}_{\text{OFFLINE}}$-Linear

The protocol is described in the $\mathcal{F}_{\text{MPC}}$-hybrid model, thus the only operation we need to specify is the **Input Function** one.

Input Function:

P_1 **Shares his Circuit/Function.**

- Player P_1 calls $(input, G_j)$ for all $j \in \{1, \ldots, g\}$.
- Players evaluate and open $[G_j] \cdot (1 - [G_j])$ for $j \in \{1, \ldots, g\}$. If any of them is not 0, players abort (since in this case P_1 has not entered a valid function).

Players Generate Randomness for inputs and outputs of EP.

- Players call $(random, \cdot)$ of $\mathcal{F}_{\text{MPC}}$ to generate shared random values for inputs $\boldsymbol{\ell} = ([\ell_1], \ldots, [\ell_{\mathsf{ow}(f)}])$ and outputs $([r_1], \ldots, [r_{\mathsf{iw}(f)}])$ of EP.
- Players call $(random, \cdot)$ of $\mathcal{F}_{\text{MPC}}$ to generate shared random values for the MAC value corresponding to inputs $\boldsymbol{t} = ([t_1], \ldots, [t_{\mathsf{ow}(f)}])$ and outputs $([s_1], \ldots, [s_{\mathsf{iw}(f)}])$ of EP.

P_1 **applies the EP to $\boldsymbol{\ell}$ and $\boldsymbol{t}$.**

- The players call KeyGen on the Enc_{Elg} functionality.
- The playes call $\mathsf{Encrypt}$ on the Enc_{Elg} functionality with the plaintexts $([\ell_1], \ldots, [\ell_{\mathsf{ow}(f)}])$ and the plaintexts $([t_1], \ldots, [t_{\mathsf{ow}(f)}])$, to obtain ciphertexts $\mathfrak{ct}_1, \ldots, \mathfrak{ct}_{\mathsf{ow}(f)}$ and $\mathfrak{ct}^{\dagger}_1, \ldots, \mathfrak{ct}^{\dagger}_{\mathsf{ow}(f)}$.
- Player P_1 applies the extended permutation to $(\mathfrak{ct}_1, \ldots, \mathfrak{ct}_{\mathsf{ow}(f)})$ and re-randomize to get $(\mathfrak{ct}'_1, \ldots, \mathfrak{ct}'_{\mathsf{ow}(f)})$, the same is done with $(\mathfrak{ct}^{\dagger}_1, \ldots, \mathfrak{ct}^{\dagger}_{\mathsf{ow}(f)})$ to obtain $(\mathfrak{ct}'^{\dagger}_1, \ldots, \mathfrak{ct}'^{\dagger}_{\mathsf{ow}(f)})$.
- Player P_1 uses the $\mathcal{ZK}_{\text{EP}}$ to prove that he has used a valid extended permutation.
- Players call the $\mathsf{Decrypt}$ on the Enc_{Elg} functionality (Figure 11) with ciphertexts $(\mathfrak{ct}'_1, \ldots, \mathfrak{ct}'_{\mathsf{ow}(f)})$ and $(\mathfrak{ct}'^{\dagger}_1, \ldots, \mathfrak{ct}'^{\dagger}_{\mathsf{ow}(f)})$ so as to obtain $([\ell_{\pi(1)}], \ldots, [\ell_{\pi(\mathsf{ow}(f))}])$ and $([t_{\pi(1)}], \ldots, [t_{\pi(\mathsf{ow}(f))}])$.

Players Compute p_i, q_i.

- For $i \in \{1, \ldots, \mathsf{iw}(f)\}$ players call $\mathcal{F}_{\text{MPC}}$ to compute:

$$[p_i] = [r_i] - [\ell_{\pi(i)}] \doteq [r_i - \ell_{\pi(i)}] \quad , \quad [q_i] = [s_i] - [t_{\pi(i)}] + p_i \cdot [K] \doteq [s_i - t_{\pi(i)} + p_i \cdot K]$$

Fig. 9: The protocol for linear implementation of the Offline Phase

The *mapping* $\pi : \{1 \dots N\} \to \{1 \dots N\}$ corresponding to a switching network SN is defined such that $\pi(j) = i$ if and only if after evaluation of SN on the N inputs, the value of the input wire i is assigned to the output wire j (assuming a standard numbering of the input/output wires). In [17] it is shown how to represent any mapping with a maximum of N inputs and outputs via a network with $O(N \cdot \log N)$ type 1 and 2 switches (We refer the reader to [17] for the details). This yields a switching network with $O(g \cdot \log g)$ switches to represent the mapping for a circuit with g gates.

High Level Description. It is possible to implement the $\mathcal{F}_{\text{OFFLINE}}$ by securely computing a circuit for the above switching network using the $\mathcal{F}_{\text{MPC}}$. But for all existing MPC that meet our requirements, this would require $O(\log g)$ rounds of interaction which is the depth of the circuit corresponding to the switching network. We show an alternative constant-round approach with similar computation and communication efficiency. It follows the same idea as the OT-based protocol of [17] where the OT is replaced with an equivalent functionality implemented using $\mathcal{F}_{\text{MPC}}$. The main challenge in our case is to achieve *active security* and in particular to ensure that P_1 cannot cheat in his local computation. We do so by checking P_1's actions using one-time MACs of the values he computes on, and allow the other parties to learn his input and proceed without him, if he is caught cheating (or aborting).

Next we give an overview of the protocol. The protocol has four main components (as described in Figure 13 and Figure 14). In the first step, P_1 converts his mapping π to selection bits for the switching network (i.e. b_is) and shares them with all players. He also shares a bit G_i indicating the function of gate i, with other players. In the second step, players generate random values for every wire in the network. P_1, based on his selection bit for the switch, learns two of the four possible "subtractions" of the random values for two output wires from those of the input wires i.e. $u_0^{\ell,i}$ and $u_1^{\ell,i}$. A similar process is performed for the t values to obtain $u_0^{t,i}$ and $u_1^{t,i}$ (Figure 10 shows this process in a diagram). These subtractions enable P_1 to transform a pair of values blinded with the random values of input wires, to the same pair of values permuted (based on the selection bit) and blinded with the random values of the output wires. All of the above can be implemented using the operations provided by the $\mathcal{F}_{\text{MPC}}$.

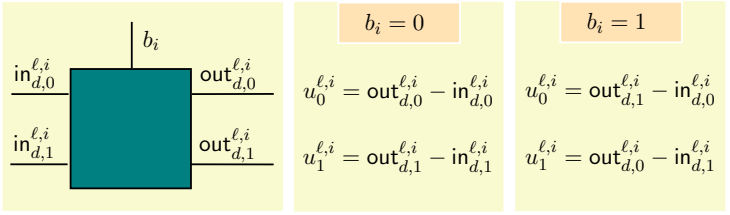

Fig. 10: The i-th switch. (superscripts: label of value subject to permute (ℓ or t), and switch index i) (subscripts: d refers to data, m refers to MAC, wire index 0 denotes the top wire in switch and 1 the bottom wire in switch)

In the third step, P_1 obtains the blinded ℓ and t values where the blinding for each is the random value for the corresponding input wire to the network (these are $h_d^{\ell,i}, h_d^{t,i}$, etc). Party P_1 can now process each switch as discussed above using the subtraction values in order to evaluate the entire network. At the end of this process, P_1 holds blinded values of the outputs of the switching network (blinded with randomness of the output wires).

In the final step, parties check that P_1 has not cheated during his evaluation, since he performed this step locally and not through the $\mathcal{F}_{\text{MPC}}$ operations. We use one-time MACs to achieve this goal. In particular, besides mapping blinded values through the network, P_1 also maps the corresponding one-time MACs (generated using the fixed-key K). This is done using a similar process described above and via the $v_j^{\ell,i}, v_j^{t,i}$ values. At the end of this process, P_1 holds one-time MACs for the blinded outputs of the switching network, in addition to the values themselves. Players then use the MPC functionality to jointly verify that the MACs indeed verify the values P_1 shared with them (i.e. $n^{\ell,i}$ and $m^{\ell,i}$ are the same, etc). As a result, P_1 can only

cheat by forging the MACs which only happens with a negligible probability. If the MACs pass, parties compute and open the "difference vectors" by subtracting the mapped ℓ and t-value vectors from the r and s-value vectors. Refer to Figure 13 and Figure 14 for more details. If one instantiates the $\mathcal{F}_{\text{MPC}}$ by SPDZ [8], which has the $m.\log(p^k)$ complexity, then our complexity would be $m\left(10(2g\log 2g - 2g + 1) + 4g\right).\log(p^k)$. Refer to Appendix Efor the proof of the following theorem.

Theorem 3. *In the $\mathcal{F}_{\text{MPC}}$-hybrid model the protocol $\mathcal{P}_{\text{Offline}}$ in Figure 13 and Figure 14 securely implements the functionality in Figure 2, with complexity $O(g \cdot \log g)$.*

6 Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant EP/I03126X, and by Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079.

The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

1. M. Abadi and J. Feigenbaum. Secure circuit evaluation. *J. Cryptology*, 2(1):1–12, 1990.
2. M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider. Secure evaluation of private linear branching programs with medical applications. In M. Backes and P. Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 424–439. Springer, 2009.
3. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.
4. J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. Privacy-preserving remote diagnostics. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 498–507. ACM, 2007.
5. D. Chaum and T. P. Pedersen. Wallet databases with observers. In *CRYPTO*, pages 89–105, 1992.
6. R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Y. Desmedt, editor, *Advances in Cryptology - CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer Berlin Heidelberg, 1994.
7. I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09*, Irvine, pages 160–179, Berlin, Heidelberg, 2009. Springer-Verlag.
8. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [25], pages 643–662.
9. I. Damgård and S. Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *Theory of Cryptography*, pages 621–641. Springer, 2013.
10. J. Furukawa. Efficient and verifiable shuffling and shuffle-decryption. *IEICE Transactions*, 88-A(1):172–188, 2005.
11. J. Furukawa and K. Sako. An efficient scheme for proving a shuffle. In J. Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 368–387. Springer Berlin Heidelberg, 2001.
12. R. Gennaro, C. Hazay, and J. S. Sorensen. Text search protocols with simulation based security. In P. Q. Nguyen and D. Pointcheval, editors, *Public Key Cryptography*, volume 6056 of *Lecture Notes in Computer Science*, pages 332–350. Springer, 2010.
13. C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. In *Public Key Cryptography*, pages 312–331, 2010.
14. Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. In S. P. Vadhan, editor, *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 575–594. Springer, 2007.

15. J. Katz and L. Malka. Constant-round private function evaluation with linear complexity. In D. H. Lee and X. Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 556–571. Springer, 2011.
16. V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In G. Tsudik, editor, *Financial Cryptography*, volume 5143 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2008.
17. P. Mohassel and S. Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 557–574. Springer, 2013.
18. C. A. Neff. A verifiable secret shuffle and its application to e-voting. In M. K. Reiter and P. Samarati, editors, *ACM Conference on Computer and Communications Security*, pages 116–125. ACM, 2001.
19. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [25], pages 681–700.
20. S. Niksefat, B. Sadeghiyan, P. Mohassel, and S. Sadeghian. Zids: A privacy-preserving intrusion detection system using secure two-party computation protocols. *The Computer Journal*, 2013.
21. R. Ostrovsky, A. Paskin-Cherniavsky, and B. Paskin-Cherniavsky. Maliciously circuit-private fhe. Cryptology ePrint Archive, Report 2013/307, 2013. `http://eprint.iacr.org/`.
22. A. Paus, A.-R. Sadeghi, and T. Schneider. Practical secure evaluation of semi-private functions. In M. Abdalla, D. Pointcheval, P.-A. Fouque, and D. Vergnaud, editors, *ACNS*, volume 5536 of *Lecture Notes in Computer Science*, pages 89–106, 2009.
23. R. Raz. Elusive functions and lower bounds for arithmetic circuits. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 711–720, New York, NY, USA, 2008. ACM.
24. A.-R. Sadeghi and T. Schneider. Generalized universal circuits for secure evaluation of private functions with application to data classification. In P. J. Lee and J. H. Cheon, editors, *ICISC*, volume 5461 of *Lecture Notes in Computer Science*, pages 336–353. Springer, 2008.
25. R. Safavi-Naini and R. Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.
26. L. Valiant. Universal circuits (preliminary report). In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 196–203. ACM, 1976.

A Instantiating Shared Encryption/Decryption

Recall our messages are elements in $\mathbf{F}_{p^k}$ and we aim to work in an elliptic curve group of prime order (to ensure DDH holds in the whole group). We therefore consider the finite field $\mathbf{F}_{p^{2k}} = \mathbf{F}_{p^k}[\theta]$, and consider an elliptic curve $E(\mathbf{F}_{p^{2k}})$ of prime order q with generator P. Let the curve be given by the equation $Y^2 = X^3 + A \cdot X + B$ where $A, B \in \mathbf{F}_{p^{2k}}$. To encrypt an element $m \in \mathbf{F}_{p^k}$ we map elements of $\mathbf{F}_{p^k}$ to elliptic curve points as follows: We pick a random $r \in \mathbf{F}_{p^k}$ and set $x = m + r \cdot \theta$. If $t = x^3 + A \cdot x + B$ is a square (which can be tested by checking if $t^{(p^{2k}-1)/2} = 1$), we extract the square root y (by the Tonelli-Shanks algorithm) and return $M = (x, y)$, otherwise we pick another r and repeat the operation. We expect this process to terminate after two steps on average.

Given M we can encrypt it by selecting $k \in \mathcal{Z}_q$ and computing $(C_1, C_2) = (k \cdot P, M + k \cdot Q)$ where $Q = \mathfrak{sk} \cdot P$ is the public key corresponding to the secret key $\mathfrak{sk}$. The decryption can be obtained via $C_2 - \mathfrak{sk} \cdot C_1$, and then simply taking the x-coordinate as $x_0 + x_1 \cdot \theta$ and returning x_0.

We need to perform the encryption however on values which are shared via the $\mathcal{F}_{\text{MPC}}$ functionality, and decrypt to obtain values which are shared via the $\mathcal{F}_{\text{MPC}}$ functionality. We first note that since the $\mathcal{F}_{\text{MPC}}$ functionality can evaluate arithmetic circuits over $\mathbf{F}_{p^k}$ it can also evaluate circuits over $\mathbf{F}_{p^{2k}}$; so for ease of exposition we will assume that $\mathcal{F}_{\text{MPC}}$ is defined over $\mathbf{F}_{p^{2k}}$. We can therefore define the functionality Enc_{Elg} given in Figure 11 in the $\mathcal{F}_{\text{MPC}}$-hybrid model. To ease notation we let $[P]$ denote a sharing of an elliptic curve point P in the $\mathcal{F}_{\text{MPC}}$ functionality in what follows. To save space we have included the protocol to implement $\mathcal{F}_{\text{MPC}}$ within the description of the functionality itself.

Functionality Enc_{Elg}

KeyGen: This generates the public key for the ElGamal encryption, given a shared secret key. The secret key is stored as shared bits for convenience, i.e. $[\mathfrak{sk}] = \sum [\mathfrak{sk}_i] \cdot 2^i$.

1. Player i calls $(input, P_i, \mathfrak{sk}_{i,j}, x_{i,j})$ for $j = 0, \ldots, \log_2 q$ and randomly selected $x_{i,j} \in \{0,1\}$ chosen by player i.
2. Define $[Q]$ as the sharing of the point at infinity.
3. This step forms $[\mathfrak{sk}_i] = \bigoplus [\mathfrak{sk}_{j,i}]$ and $[Q] = \sum [\mathfrak{sk}_i] \cdot 2^i \cdot P$, and ensures that the players input values in the first step are in $\{0,1\}$. We perform this step by executing, for $i = 0, \ldots, \log_2 q$,
   - $[\mathfrak{sk}_i] = [\mathfrak{sk}_{0,i}]$
   - For $j = 2, \ldots, n$ do $[\mathfrak{sk}_i] = -2 \cdot [\mathfrak{sk}_i] \cdot [\mathfrak{sk}_{j,i}] + [\mathfrak{sk}_i] + [\mathfrak{sk}_{j,i}]$, using the MPC functionality.
   - Compute $[t_i] = [\mathfrak{sk}_i] \cdot ([\mathfrak{sk}_i] - 1)$, again using the MPC functionality.
   - Call $(output, t_i)$ to open $[t_i]$, if the value is not zero then restart.
   - Execute $[Q] = [Q] + [\mathfrak{sk}_i] \cdot 2^i \cdot P$. Here we use the $\mathcal{F}_{\text{MPC}}$ functionality to evaluate the conditional elliptic curve addition.
4. The players call $(output, Q)$ to open $[Q]$.

Encrypt: This takes an input message $[m]$ where $m \in \mathbf{F}_{p^k}$ and outputs an ElGamal ciphertext (C_1, C_2).

1. Using a method similar to that for KeyGen above the players generate sharings of bits $[k_i]$ for $i = 0, \ldots, \log_2 q$ and then evaluate $[kP]$ and $[kQ]$ for k the integer with bit representation given by the shared bits $[k_i]$.
2. The players call $(random, r)$.
3. The players execute $[x] = [m] + \theta \cdot [r]$.
4. The players execute $[t] = [x^3] + A \cdot [x] + B$.
5. The players compute $[s] = [t^{(p^{2k}-1)/2}]$ and call $(output, s)$ to open $[s]$.
6. If $s \neq 1$ then goto step 2.
7. The players execute the Tonelli-Shanks algorithm to extract the square root $[y]$ of $[t]$ using the $\mathcal{F}_{\text{MPC}}$ functionality.
8. The players execute $[G] = ([x], [y]) + [kQ]$.
9. The players call $(output, \cdot)$ on the x and y coordinates of $[kP]$ and $[G]$ so as to obtain C_1 and C_2.

Decrypt: Obtain the sharing of the message $[m]$ corresponding to ciphertext (C_1, C_2).

1. The players execute using $\mathcal{F}_{\text{MPC}}$ the operations corresponding to $[G] = C_2 - \sum_{i=0}^{\log_2 q} [\mathfrak{sk}_i] \cdot 2^i \cdot C_1$.
2. Consider $[G]$ as having x-coordinate $[m] + \theta \cdot [m']$ and output $[m]$.

Fig. 11: Elgamal Functionality

B An Incomplete Attempt to Extend Existing Proofs of Shuffle

In this section we explain our attempt at extending the existing proofs of shuffle to extended permutation. Current available solutions are following two main ideas: The first group started by Furukawa and Sako [11] represents the permutation by a permutation matrix and then proves using ZK that it is a valid permutation and is used in computation. The second group started by Neff [18] uses the property of polynomials of being identical under permutation of their roots.

In the second group, it is not obvious how it is possible to handle variant number of repetitions for each root. On the other hand it is possible to represent an EP using a matrix.

We turn to modifying the method of Furukawa and Sako [11] (and the later work by Furukawa [10]), to check an extended permutation. We only describe the general idea, for more details concerning our modifications we refer the reader to the original paper [11]. In their protocol they use the matrix representation of permutation and prove that the matrix used for computation of outputs is a valid permutation (i.e. there is exactly one non-zero element one in each row and each column). For our purpose of extended permutation, it is only enough to show that there is exactly one non-zero element, *one* in each column of matrix. Theorem 4 shows the conditions for a matrix to be an extended permutation.

Theorem 4. *A matrix $(A_{ij})_{i,j=1,\dots,n)}$ is an extended permutation if and only if, for all i, j and k, the following conditions hold:*

$$\sum_{h=1}^{n} A_{hi} = 1 \pmod{q} \tag{1}$$

For all $i, j : (i \neq j)$

$$\sum_{h=1}^{n} A_{ih} \cdot A_{jh} = 0 \pmod{q} \tag{2}$$

For all $i, j, k : \neg(i = j = k)$

$$\sum_{h=1}^{n} A_{ih} \cdot A_{jh} \cdot A_{kh} = 0 \pmod{q} \tag{3}$$

Proof (sketch). The first condition implies that there is at least one non-zero element in each column. Using the similar argument to [11], for $i \neq j$, the second and third conditions imply that the number of non-zero elements in each column is at most one. From first condition, this non-zero element should be one.

This theorem allows us to adapt the zero-knowledge protocol given in [11]. The main challenge in their protocol is to give proof for the conditions of equations 2,3. We assume that the prover has applied the extended permutation to the ciphertexts $(\mathfrak{ct}_1, \dots, \mathfrak{ct}_n)$, where $\mathfrak{ct}_i = (C_1^{(i)}, C_2^{(i)})$. The prover obtains a re-randomized $(\mathfrak{ct}'_1, \dots, \mathfrak{ct}'_n)$, where $\mathfrak{ct}'_i = (C_1'^{(i)}, C_2'^{(i)})$ and $C_1'^{(i)} = k'_i \cdot P + C_1^{(\pi(i))}$, $C_2'^{(i)} = k'_i \cdot Q + C_2^{(\pi(i))}$.

To prove the condition in equation 2, we have to show that given $\{C_1^{(i)}\}$ and $\{C_1'^{(i)}\}$, the prover knows k'_i and A_{ij} such that:

$$C_1'^{(i)} = k'_i \cdot P + \sum_{j=0}^{n} A_{ij} \cdot C_1^{(j)} \quad \text{and} \quad \sum_{h=1}^{n} A_{ih} \cdot A_{jh} = 0.$$

In [11] they suggest to issue values s and s_i as a respond to challenge c_j and let the verifier check two conditions. We adjust s_i for our modified scenario such that s_i^2 generates the condition of equation 2:

$$s_i = \sum_{j=1}^{n} A_{ji} c_j \pmod{q},$$

At this point it is not obvious how to issue s, and define the second verification equation considering the modified s_i.

C Proof of Protocol $\mathcal{P}_{\text{Offline}}$-Linear

We construct a simulator $\mathcal{S}_{\text{OFFLINE}}$ such that a poly-time environment $\mathcal{Z}$ cannot distinguish between the real protocol system and the ideal. We assume here static, active corruption. The simulator runs a copy of the protocol given in Figure 9, which simulates the ideal functionality given in Figure 2. It relays messages between parties/$\mathcal{F}_{\text{MPC}}$ and $\mathcal{Z}$, such that $\mathcal{Z}$ will see the same interface as when interacting with a real protocol. The specification of the simulator $\mathcal{S}_{\text{OFFLINE}}$ is presented in Figure 12.

Simulator $\mathcal{S}_{\text{OFFLINE}}$

The protocol is described in the $\mathcal{F}_{\text{MPC}}$-hybrid model, thus we only need to specify the simulator for the **Input Function** one. Let's denote the set of corrupted parties by $C \subset \{P_1, \ldots, P_N\}$.

Input Function:

P_1 Shares his Circuit/Function.

- $P_1 \in C$:
  - Simulator $\mathcal{S}_{\text{OFFLINE}}$ evaluates $[G_j] \cdot (1 - [G_j])$ for $j \in \{1, \ldots, g\}$. If any of them is not 0, simulator abort (since in this case P_1 has not entered a valid function).
- $P_1 \notin C$:
  - Simulator $\mathcal{S}_{\text{OFFLINE}}$ generates a random circuit with g gates G'_j for all $j \in \{1, \ldots, g\}$ and finds its corresponding mapping π'.
  - $\mathcal{S}_{\text{OFFLINE}}$ calls $(input, G'_j)$ for all $j \in \{1, \ldots, g\}$.
  - Simulator $\mathcal{S}_{\text{OFFLINE}}$ evaluates $[G'_j] \cdot (1 - [G'_j])$ for $j \in \{1, \ldots, g\}$.

Players Generate Randomness for inputs and outputs of EP.

- Simulator $\mathcal{S}_{\text{OFFLINE}}$ follows the protocol honestly.

P_1 **Applies the EP to $s\ell$ and st.**

- $P_1 \in C$:
  - Simulator $\mathcal{S}_{\text{OFFLINE}}$ randomly generates an extended permutation π' and sends it to $\mathcal{ZK}_{\text{EP}}$ ideal functionality. Simulator aborts if any of players aborts.
- $P_1 \notin C$:
  - Simulator follows the protocol honestly and sends $s\ell$ and st to Enc_{Elg} ideal functionality.
  - Simulator $\mathcal{S}_{\text{OFFLINE}}$ waits for P_1 to broadcast π to $\mathcal{ZK}_{\text{EP}}$ ideal functionality, he then sends π to ideal functionality $\mathcal{F}_{\text{OFFLINE}}$. Simulator aborts if any of players aborts.
  - Simulator follows the protocol honestly.

Players Check P_1's Work and Compute p_i, q_i.

- Simulator $\mathcal{S}_{\text{OFFLINE}}$ follows the protocol honestly.

Fig. 12: Simulator $\mathcal{S}_{\text{OFFLINE}}$

To see that the simulated and real processes cannot be distinguished, we will show that the view of the environment in the ideal process is statistically indistinguishable from the view in the real process. This view consists of the corrupt players' view of the protocol execution as well as the inputs and outputs of honest players.

The view of adversaries $C - \{P_1\}$, includes the share of G_i, the share of random values for inputs and outputs of EP, $([s\ell_1], \ldots, [s\ell_{\mathsf{ow}(f)}])$, $([sr_1], \ldots, [sr_{\mathsf{iw}(f)}])$, $([st_1], \ldots, [st_{\mathsf{ow}(f)}])$, $([ss_1], \ldots, [ss_{\mathsf{iw}(f)}])$, $([s\ell_{\pi(1)}], \ldots, [s\ell_{\pi(\mathsf{ow}(f))}])$, $([st_{\pi(1)}], \ldots, [st_{\pi(\mathsf{ow}(f))}])$, $(s\mathfrak{ct}_1, \ldots, s\mathfrak{ct}_{\mathsf{ow}(f)})$, $(s\mathfrak{ct}'_1, \ldots, s\mathfrak{ct}'_{\mathsf{ow}(f)})$, $(s\mathfrak{ct}^{\dagger}_1, \ldots, s\mathfrak{ct}^{\dagger}_{\mathsf{ow}(f)})$, $(s\mathfrak{ct}'^{\dagger}_1, \ldots, s\mathfrak{ct}'^{\dagger}_{\mathsf{ow}(f)})$, and finally, p_i, q_i. The shared values all look random and therefore are indistinguishable between ideal and real execution. $(s\mathfrak{ct}_1, \ldots, s\mathfrak{ct}_{\mathsf{ow}(f)})$ and $(s\mathfrak{ct}^{\dagger}_1, \ldots, s\mathfrak{ct}^{\dagger}_{\mathsf{ow}(f)})$ are ElGamal encryptions under shared secret key, and therefore are indistinguishable from real execution. $(s\mathfrak{ct}'_1, \ldots, s\mathfrak{ct}'_{\mathsf{ow}(f)})$ and $(s\mathfrak{ct}'^{\dagger}_1, \ldots, s\mathfrak{ct}'^{\dagger}_{\mathsf{ow}(f)})$ are valid re-randomization of ElGamal ciphertexts if protocol does not abort due to $\mathcal{ZK}_{\text{EP}}$ verification. $([s\ell_{\pi(1)}], \ldots, [s\ell_{\pi(\mathsf{ow}(f))}])$, $([st_{\pi(1)}], \ldots, [st_{\pi(\mathsf{ow}(f))}])$ are freshly new shares generated by Enc_{Elg} protocol. The final result p_i, q_i is computed as a result of two shared random values, and therefore has a uniform distribution in both ideal and

real executions. The view of malicious P_1, is the same view as other malicious players. The shared values all have uniform distribution. In the ideal functionality we also have a uniform distribution, and as a result ideal and real executions are indistinguishable to the environment $\mathcal{Z}$.

■

D Complete Description of Protocol $\mathcal{P}_{\text{Offline}}$

See Figure 13 and Figure 14 for the description of protocol $\mathcal{P}_{\text{OFFLINE}}$.

E Proof of Theorem 3

We construct a simulator $\mathcal{S}_{\text{OFFLINE}}$ such that a poly-time environment $\mathcal{Z}$ cannot distinguish between the real protocol system and the ideal. We assume here static, active corruption. The simulator runs a copy of the protocol given in Figure 13 and Figure 14, which simulates the ideal functionality given in Figure 2. It relays messages between parties/$\mathcal{F}_{\text{MPC}}$ and $\mathcal{Z}$, such that $\mathcal{Z}$ will see the same interface as when interacting with a real protocol. The specification of the simulator $\mathcal{S}_{\text{OFFLINE}}$ is presented in Figure 15.

To see that the simulated and real processes cannot be distinguished, we will show that the view of the environment in the ideal process is statistically indistinguishable from the view in the real process. This view consists of the corrupt players' view of the protocol execution as well as the inputs and outputs of honest players.

The view of adversaries $C - \{P_1\}$, includes the share of b_i, G_i, the share of wires' random values, $h_d^{\ell,i}, h_d^{t,i}$, $[d^{\ell,i}], [d^{t,i}]$ and finally, $n^{\ell,i}$ and p_i, q_i. The shared values all look random and therefore are indistinguishable between ideal and real execution. The final result p_i, q_i is computed as a result of two shared random values, and therefore has a uniform distribution in both ideal and real execution. The values $h_d^{\ell,i}, h_d^{t,i}$ are blinded by shared values ℓ and t respectively and have uniform distribution.

The view of malicious P_1, includes the share of b_i, G_i, share of wires' random values, $h_d^{\ell,i}, h_d^{t,i}$, $d^{\ell,i}, d^{t,i}$ and finally, $n^{\ell,i}$ and p_i, q_i. P_1 has the same view as other malicious players except for the $d^{\ell,i}, d^{t,i}$ values that he has computed . It only remains to show that $d^{\ell,i}, d^{t,i}$ have uniform distribution for a malicious P_1 and checks are guaranteeing the correctness of his computation. Observe that $h_d^{\ell,i}$ is blinded using random value of input wires which is shared and therefore acts as a one-time pad, and as P_1 does the evaluation the distribution remains uniform as he continues. Using the similar argument, $d^{t,i}$ has a uniform distribution due to $h_d^{t,i}$. In the ideal functionality we also have a uniform distribution, and as a result ideal and real are indistinguishable to the environment $\mathcal{Z}$. In the final phase players check the P_1's computation. Player P_1 cheating means he has not calculated $d^{\ell,i}, d^{t,i}$ correctly. For him to be successful, he has to somehow adjust $n^{\ell,i}$ and $m^{\ell,i}$ to be equal. Any modification is prevented by the fact that since he does not know the key K, it acts as a one-time MAC and therefore he can not adjust his share $[\mathsf{out}_{m,j}^{\ell,\lceil i/2 \rceil}]$ to make the equality hold. The probability of him getting away with it is equal to him guessing K and hence exponentially small in the length of K. It follows that with overwhelming probability after the check P_1's computation has been done correctly. If any check fails the simulator aborts and stop.

■

F Proof of Protocol $\mathcal{P}_{\text{Online}}$

We construct a simulator $\mathcal{S}_{\text{ONLINE}}$ such that a poly-time environment $\mathcal{Z}$ cannot distinguish between the real protocol system and the ideal. We assume here static, active corruption. The simulator runs a copy of the protocol $\mathcal{P}_{\text{ONLINE}}$ given in Figure 6, which simulates the ideal functionalities given in Figure 4. It relays messages between parties/$\mathcal{F}_{\text{OFFLINE}}$ and $\mathcal{Z}$, such that $\mathcal{Z}$ will see the same interface as when interacting with a real protocol. The specification of the simulator $\mathcal{S}_{\text{ONLINE}}$ is presented in Figure 16.

Protocol $\mathcal{P}_{\text{OFFLINE}}$ Part I

The protocol is described in the $\mathcal{F}_{\text{MPC}}$-hybrid model, thus the only operation we need to specify is the **Input Function** one.

Input Function:

P_1 **Shares his Circuit/Function.**

- P_1 determines a vector of selection bits $(b_1, \ldots, b_N)$ corresponding to the switching network representing the mapping π. Note that the switching network has $\mathsf{ow}(f)$ input wires and $\mathsf{iw}(f)$ output wires.
- Player P_1 calls $(input, b_i)$ for all $i \in \{1, \ldots, N\}$.
- Player P_1 calls $(input, G_j)$ for all $j \in \{1, \ldots, g\}$.
- Players evaluate and open $[b_i] \cdot (1 - [b_i])$ for all $i \in \{1, \ldots, N\}$ and similarly for $[G_j] \cdot (1 - [G_j])$ for $j \in \{1, \ldots, g\}$. If any of them is not 0, players abort (since in this case P_1 has not entered a valid function).

Players Generate Randomness for the Switching Network.

- The players call $(random, K)$ of $\mathcal{F}_{\text{MPC}}$.
- Players call $(random, \cdot)$ of $\mathcal{F}_{\text{MPC}}$ to generate two pairs of shared random values for each wire in the switching network; one pair is used to map the ℓ values and another to map the t values (recall each value $j \in \{1, \ldots, \mathsf{ow}(f)\}$ has a value ℓ_j and t_j).
  Let us denote the two shared random pairs for the jth input wire ($j \in \{0, 1\}$) of the ith switch by $([\mathsf{in}_{d,j}^{\ell,i}], [\mathsf{in}_{m,j}^{\ell,i}])$ and $([\mathsf{in}_{d,j}^{t,i}], [\mathsf{in}_{m,j}^{t,i}])$, and the pairs for its two output wires by $([\mathsf{out}_{d,j}^{\ell,i}], [\mathsf{out}_{m,j}^{\ell,i}])$ and $([\mathsf{out}_{m,j}^{t,i}], [\mathsf{out}_{m,j}^{t,i}])$. (The d subscript means the random value is used to process *data* (actual wire values) while the m subscripts means the random value is used for the corresponding *macs*. The subscript $j \in \{0, 1\}$ determines which wire of the switch the value corresponds to. 0 means the the top wire while 1 denotes the bottom wire.)
- Then, for each switch i in the network players perform the following (in parallel):
  - The players call $\mathcal{F}_{\text{MPC}}$ to evaluate and open the following for $j \in \{0, 1\}$ (the following corresponds to switch type 1 but a similar approach works for type 2 switches)

$$
\begin{aligned}
[u_j^{\ell,i}] &= (1 - [b_i]) \cdot ([\mathsf{out}_{d,j}^{\ell,i}] - [\mathsf{in}_{d,j}^{\ell,i}]) + [b_i] \cdot ([\mathsf{out}_{d,1-j}^{\ell,i}] - [\mathsf{in}_{d,j}^{\ell,i}]),\\
[u_j^{t,i}] &= (1 - [b_i]) \cdot ([\mathsf{out}_{d,j}^{t,i}] - [\mathsf{in}_{d,j}^{t,i}]) + [b_i] \cdot ([\mathsf{out}_{d,1-j}^{t,i}] - [\mathsf{in}_{d,j}^{t,i}]),\\
[v_j^{\ell,i}] &= (1 - [b_i]) \cdot ([\mathsf{out}_{m,j}^{\ell,i}] - [\mathsf{in}_{m,j}^{\ell,i}]) + [b_i] \cdot ([\mathsf{out}_{m,1-j}^{\ell,i}] - [\mathsf{in}_{m,j}^{\ell,i}])\\
&\quad + u_j^{\ell,i} \cdot [K],\\
[v_j^{t,i}] &= (1 - [b_i]) \cdot ([\mathsf{out}_{m,j}^{t,i}] - [\mathsf{in}_{m,j}^{t,i}]) + [b_i] \cdot ([\mathsf{out}_{m,1-j}^{t,i}] - [\mathsf{in}_{m,j}^{t,i}])\\
&\quad + u_j^{t,i} \cdot [K].
\end{aligned}
$$

  Note, the final two equations can be evaluated using the open values of $u_j^{\ell,i}$ and $u_j^{t,i}$.
- For $i \in \{1, \ldots, \mathsf{ow}(f)\}$ players call $\mathcal{F}_{\text{MPC}}$ to evaluate and open (let $j = i \mod 2$)

$$
\begin{aligned}
[h_d^{\ell,i}] &= [\ell_i] + [\mathsf{in}_{d,j}^{\ell,\lceil i/2 \rceil}], \qquad [h_m^{\ell,i}] = [\mathsf{in}_{m,j}^{\ell,\lceil i/2 \rceil}] + h_d^{\ell,i} \cdot [K],\\
[h_d^{t,i}] &= [t_i] + [\mathsf{in}_{d,j}^{t,\lceil i/2 \rceil}], \qquad [h_m^{t,i}] = [\mathsf{in}_{m,j}^{t,\lceil i/2 \rceil}] + h_d^{t,i} \cdot [K],
\end{aligned}
$$

Fig. 13: The protocol to implement the Offline Phase: Part I

Protocol $\mathcal{P}_{\text{OFFLINE}}$ Part II

P_1 Maps the ℓ and t Values Using the Above Randomness.

- For $i \in \{1, \ldots, \mathsf{iw}(f)\}$, P_1 determines the sequence of switches involved in mapping the input label $\pi(i) \in \mathsf{ow}(f)$ to the output label $i \in \mathsf{iw}(f)$. Denote the sequence of switches by $(i_1, \ldots, i_k)$, and the index of the input wire the values goes through by $j_1, \ldots, j_k$. Note that $k = O(\log N)$, $i_k = \lceil i/2 \rceil$, and $j_k = i \mod 2$.
- P_1 then computes the following d, m values and calls $(input, \cdot)$ of the $\mathcal{F}_{\text{MPC}}$ on each to store the value in the functionality (i.e. share among the parties)

$$d^{\ell,i} = h_d^{\ell,\pi(i)} + \sum_{j=1}^{k} u_{j_k}^{\ell,i_j} \doteq \ell_{\pi(i)} + \mathsf{out}_{d,j_k}^{\ell,i_k},$$

$$m^{\ell,i} = h_m^{\ell,\pi(i)} + \sum_{j=1}^{k} v_{j_k}^{\ell,i_j} \doteq \mathsf{out}_{m,j_k}^{\ell,i_k} + d^{\ell,i} \cdot K,$$

$$d^{t,i} = h_d^{t,\pi(i)} + \sum_{j=1}^{k} u_{j_k}^{t,i_j} \doteq t_{\pi(i)} + \mathsf{out}_{d,j_k}^{t,i_k},$$

$$m^{t,i} = h_m^{t,\pi(i)} + \sum_{j=1}^{k} v_{j_k}^{t,i_j} \doteq \mathsf{out}_{m,j_k}^{t,i_k} + d^{t,i} \cdot K,$$

Players Check P_1's Work and Compute p_i, q_i.

- For $i \in \{1, \ldots, \mathsf{iw}(f)\}$ players call $\mathcal{F}_{\text{MPC}}$ to compute (let $j = i \mod 2$)

$$[n^{\ell,i}] = [\mathsf{out}_{m,j}^{\ell,\lceil i/2 \rceil}] + [d^{\ell,i}] \cdot [K], \quad [n^{t,i}] = [\mathsf{out}_{m,j}^{t,\lceil i/2 \rceil}] + [d^{t,i}] \cdot [K].$$

- Parties then compute and open $[n^{\ell,i} - m^{\ell,i}]$ and $[n^{t,i} - m^{t,i}]$. If either is not 0, players call **Cheat**(1) on the $\mathcal{F}_{\text{MPC}}$ functionality. This will either abort, or return the input of P_1 (and hence the function), in the latter case the players can now proceed with evaluating the function using standard MPC and without the need for P_1 to be involved. If the opened value is zero the players compute and open

$$[p_i] = [r_i] - [d^{\ell,i}] + [\mathsf{out}_{d,j}^{\ell,\lceil i/2 \rceil}] \doteq [r_i - \ell_{\pi(i)}],$$
$$[q_i] = [s_i] - [d^{t,i}] + [\mathsf{out}_{d,j}^{t,\lceil i/2 \rceil}] + p_i \cdot [K] \doteq [s_i - t_{\pi(i)} + p_i \cdot K],$$

Fig. 14: The protocol to implement the Offline Phase: Part II

Simulator $\mathcal{S}_{\text{OFFLINE}}$

The protocol is described in the $\mathcal{F}_{\text{MPC}}$-hybrid model, thus we only need to specify the simulator for the **Input Function** one. Let's denote the set of corrupted parties by $C \subset \{P_1, \ldots, P_N\}$.

Input Function:

P_1 Shares his Circuit/Function.

- $P_1 \in C$:
  - Simulator $\mathcal{S}_{\text{OFFLINE}}$ runs the protocol honestly and then waits for P_1 to broadcast b_i for all $i \in \{1, \ldots, N\}$ and G_j for all $j \in \{1, \ldots, g\}$, he then sends them to ideal functionality $\mathcal{F}_{\text{OFFLINE}}$.
  - Simulator $\mathcal{S}_{\text{OFFLINE}}$ evaluates $b_i \cdot (1 - b_i)$ for all $i \in \{1, \ldots, N\}$ and similarly for $[G_j] \cdot (1 - [G_j])$ for $j \in \{1, \ldots, g\}$. If any of them is not 0, simulator abort (since in this case P_1 has not entered a valid function).
- $P_1 \notin C$:
  - Simulator $\mathcal{S}_{\text{OFFLINE}}$ generates a random circuit with g gates G'_j for all $j \in \{1, \ldots, g\}$ and finds its corresponding mapping π'. Then it determines a vector of selection bits $(b'_1, \ldots, b'_N)$ corresponding to the switching network representing the mapping π'.
  - $\mathcal{S}_{\text{OFFLINE}}$ calls $(input, b'_i)$ for all $i \in \{1, \ldots, N\}$.
  - $\mathcal{S}_{\text{OFFLINE}}$ calls $(input, G'_j)$ for all $j \in \{1, \ldots, g\}$.
  - Simulator $\mathcal{S}_{\text{OFFLINE}}$ evaluates $[b'_i] \cdot (1 - [b'_i])$ for all $i \in \{1, \ldots, N\}$ and similarly for $[G'_j] \cdot (1 - [G'_j])$ for $j \in \{1, \ldots, g\}$.

Players Generate Randomness for the Switching Network.

- Simulator $\mathcal{S}_{\text{OFFLINE}}$ follows the protocol honestly.

P_1 **Maps the ℓ and t Values Using the Randomness.**

- Simulator $\mathcal{S}_{\text{OFFLINE}}$ follows the protocol honestly.

Players Check P_1's Work and Compute p_i, q_i.

- Players follow the steps of protocol and simulator aborts if the checks were failed by any of players.

Fig. 15: Simulator $\mathcal{S}_{\text{OFFLINE}}$

Simulator $\mathcal{S}_{\text{ONLINE}}$

The protocol is described in the $\mathcal{F}_{\text{OFFLINE}}$-hybrid model.

Input Function: If $P_1 \notin C$, simulator generates a random circuit with g gates and corresponding mapping π', and follows the protocol honestly. If $P_1 \in C$, simulator $\mathcal{S}_{\text{ONLINE}}$ runs the protocol honestly and then waits for P_1 to broadcast π and f, he then sends them to ideal functionality $\mathcal{F}_{\text{ONLINE}}$.

Input Data: If $P_i \notin C$, simulator generates a dummy input x_i' and follows the steps of protocol honestly. If $P_i \in C$, simulator runs the protocol honestly and waits for them to send their input to $\mathcal{F}_{\text{OFFLINE}}$, he then sends them to $\mathcal{F}_{\text{ONLINE}}$ ideal functionality.

Output: Simulator follows the protocol steps honestly. For $P_i \notin C$

- **Preparing Inputs to the Circuit:**
  - Simulator follows the steps of protocol honestly.
- **Evaluating the Circuit:** For every gate $1 \leq i \leq g$ in the circuit players execute the following (here we assume that the gates are indexed in the same topological order P_1 chose to determine π):
  - P_1 **Prepares the Two Inputs for Gate** i**.**
    - Simulator follows the steps of protocol honestly.
  - **Players Check** P_1**'s Input Preparation.**
    - Simulator follows the steps of protocol honestly and aborts if the checks are failed.
  - **Players Jointly Evaluate Gate** i**.**
    - The players store the value $[y_{i_j}] = d_{i_j} - [r_{i_j}]$ in the $\mathcal{F}_{\text{MPC}}$ functionality.
    - The $\mathcal{F}_{\text{MPC}}$ functionality is then executed so as to compute the output of the gate as

$$[z_i] = (1 - [G_i]) \cdot ([y_{i_0}] + [y_{i_1}]) + [G_i] \cdot [y_{i_0}] \cdot [y_{i_1}].$$

    - Note that the outgoing wire label corresponding to the output wire of the ith gate is $j = n + i$ (the first n outgoing wires are input wires, hence output wire of the ith gate is indexed $n + i$) so we just relabel $[z_i]$ to $[z_j]$.
    - The players compute via the MPC functionality $[u_j] = [z_j] + [\ell_j]$.
    - The players call (Output, u_j) so as to obtain u_j.
    - The players then compute via the MPC functionality

$$[v_j] = [t_j] + u_j \cdot [K] = [t_j + (z_j + \ell_j) \cdot K].$$

    - The players call (Output, v_j) so as to obtain v_j. If j is the output wire, simulator adjusts his share of output in the ideal execution to make the output consistent with the shares of honest parties as follows: suppose the output of that wire using the dummy values is z_i and the output returned by the $\mathcal{F}_{\text{ONLINE}}$ ideal functionality is z_i', he then adds $z_i - z_i'$ to the share of adversary $[z_i']$ in the ideal execution.

Fig. 16: The Protocol for implementing PFE

To see that the simulated and real processes cannot be distinguished, we will show that the view of the environment in the ideal process is statistically indistinguishable from the view in the real process. This view consists of the corrupt players' view of the protocol execution as well as the inputs and outputs of honest players. The view of adversary includes $[u_i], [v_i]$, d_{i_j}, m_{i_j}, n_{i_j}, $[z_i]$ and if i is the index of output wire, z_i. The shared values all look random and therefore are indistinguishable between ideal and real execution.

We next show that d_{i_j}, m_{i_j} have uniform distribution. Observe that u_i is blinded using the random value of input wires which is shared and therefore acts as a one-time pad, and as P_1 prepares the two inputs, it maintains the uniform distribution. Furthermore, p_{i_j} also has uniform distribution from the security of offline protocol. The value s_{i_j} acts as a one-time pad which is shared between the players and therefore, m_{i_j} has a uniform distribution. In the ideal functionality we also have a uniform distribution, and as a result ideal and real are indistinguishable to the environment $\mathcal{Z}$.

For a malicious P_1, the distributions are the same, but we have to make sure that he has performed the input preparation correctly. In the next phase players check the P_1's computation. Player P_1 cheating means he has not calculated d_{i_j}, m_{i_j} correctly. For him to be successful, he has to somehow adjust n_{i_j} and m_{i_j} to be equal. He only has a option to adjust d_{i_j} and his share of $[S_{i_j}]$ to make the equality hold. Since he does not know K, the value $d_{i_j} \cdot K$ has a uniform distribution, and therefore the probability of him modifying $[S_{i_j}]$ to make the equality hold is equivalent to guessing K and hence exponentially small in length of K. It follows that with overwhelming probability after the check the P_1's computation has been done correctly. If any check fails the simulator aborts and stop.

The final result z_i is a secret shared value and as result has a uniform distribution. For the output wires, players open their share, and z_i is learnt by all parties. In order to make the distribution of outputs indistinguishable, the simulator has to modify his share of z_i in the ideal execution. He is able to do so and produce the exact same output for the ideal execution as described in Figure 16. This completes the proof.

■

"Ooh Aah... Just a Little Bit" : A small amount of side channel can go a long way

Naomi Benger[1], Joop van de Pol[2], Nigel P. Smart[2], and Yuval Yarom[1]

[1] School of Computer Science, The University of Adelaide, Australia.
mail.for.minnie@gmail.com,yval@cs.adelaide.edu.au
[2] Dept. Computer Science, University of Bristol, United Kingdom.
joop.vandepol@bristol.ac.uk,nigel@cs.bris.ac.uk

Abstract. We apply the FLUSH+RELOAD side-channel attack based on cache hits/misses to extract a small amount of data from OpenSSL ECDSA signature requests. We then apply a "standard" lattice technique to extract the private key, but unlike previous attacks we are able to make use of the side-channel information from almost all of the observed executions. This means we obtain private key recovery by observing a relatively small number of executions, and by expending a relatively small amount of post-processing via lattice reduction. We demonstrate our analysis via experiments using the curve **secp256k1** used in the Bitcoin protocol. In particular we show that with as little as 200 signatures we are able to achieve a reasonable level of success in recovering the secret key for a 256-bit curve. This is significantly better than prior methods of applying lattice reduction techniques to similar side channel information.

1 Introduction

One important task of cryptographic research is to analyze cryptographic implementations for potential security flaws. This aspect has a long tradition, and the most well known of this line of research has been the understanding of side-channels obtained by power analysis, which followed from the initial work of Kocher and others [24]. More recently work in this area has shifted to looking at side-channels in software implementations, the most successful of which has been the exploitation of cache-timing attacks, introduced in 2002 [34]. In this work we examine the use of spy-processes on the OpenSSL implementation of the ECDSA algorithm.

OpenSSL [33] is an open source tool kit for the implementation of cryptographic protocols. The library of functions, implemented using C, is often used for the implementation of Secure Sockets Layer and Transport Layer Security protocols and has also been used to implement OpenPGP and other cryptographic standards. The library includes cryptographic functions for use in Elliptic Curve Cryptography (ECC), and in particular ECDSA. In particular we will examine the application of the FLUSH+RELOAD attack, first proposed by Yarom and Falkner [43], then adapted to the case of OpenSSL's implementation of ECDSA over binary fields by Yarom and Benger [42], running on X86 processor architecture. We exploit a property of the Intel implementation of the X86 and X86_64 processor architectures using the FLUSH+RELOAD cache side-channel attack [42, 43] to partially recover the ephemeral key used in ECDSA.

In Yarom and Benger [42] the case of characteristic two fields was considered, but the algorithms used by OpenSSL in the characteristic two and prime characteristic cases are very different. In particular for the case of prime fields one needs to perform a post-processing of the side-channel information using cryptanalysis of lattices. We adopt a standard technique [23, 32] to perform this last step, but in a manner which enables us to recover the underlying secret with few protocol execution runs. This is achieved by using as much information obtained in the FLUSH+RELOAD step as possible in the subsequent lattice step.

We illustrate the effectiveness of the attack by recovering the secret key with a very high probability using only a small number of signatures. After this, we are able to forge unlimited signatures under the hidden secret key. The results of this attack are not limited to ECDSA but have implications for many other cryptographic protocols implemented using OpenSSL for which the scalar multiplication is performed using a sliding window and the scalar is intended to remain secret.

Related Work: Microarchitectural side-channel attacks have been used against a number of implementations of cryptosystems. These attacks often target the L1 cache level [1, 2, 5, 10, 13, 14, 39, 44] or the branch prediction buffer [3, 4]. The use of these components is limited to a single execution core. Consequently, the spy program and the victim must execute on the same execution core of the processor. Unlike these attacks, the FLUSH+RELOAD attack we use targets the last level cache (LLC). As the LLC is shared between cores, the attack can be mounted between different cores.

The attack used by Gullasch et al. [22] against AES, is very similar to FLUSH+RELOAD. The attack, however, requires the interleaving of spy and victim execution on the same processor core, which is achieved by relying on a scheduler bug to interrupt the victim and gain control of the core on which it executes. Furthermore, the Gullasch et al. attack results in a large number of false positives, requiring the use of a neural network to filter the results.

In [43], Yarom and Falkner first describe the FLUSH+RELOAD attack and use it to snoop on the square-and-multiply exponentiation in the GnuPG implementation of RSA and thus retrieve the RSA secret key from the GnuPG decryption step. The OpenSSL (characteristic two) implementation of ECDSA was also shown to be vulnerable to the FLUSH+RELOAD attack [42]; around 95% of the ephemeral private key was recovered when the Montgomery ladder was used for the scalar multiplication step. The full ephemeral private key was then recovered at very small cost using a Baby-Step-Giant-Step (BSGS) algorithm. Knowledge of the ephemeral private key leads to recovery of the signer's private key, thus fully breaking the ECDSA implementation using only one signature.

One issue hindering the extension of the attack to implementations using the sliding window method for scalar multiplications instead of the Montgomery ladder is that only a lower proportion of the bits of the ephemeral private key can be recovered so the BSGS reconstruction becomes infeasible. It is to extend the FLUSH+RELOAD attack to implementations which use sliding window exponentiation methods that this paper is addressed.

Suppose we take a single ECDLP instance, and we have obtained partial information about the discrete logarithm. In [21, 28, 38] techniques are presented which reduce the search space for the underlying discrete logarithm when various types of partial information is revealed. These methods work quite well when the information leaked is considerable for the single discrete logarithm instance; as for example evidenced by the side-channel attack of [42] on the Montgomery ladder. However, in our situation a different approach needs to be taken.

Similar to several past works, e.g. [10, 11, 29], we will exploit a well known property of ECDSA, that if a small amount of information about each ephemeral key in each signature leaks, for a number of signatures, then one can recover the underlying secret using a lattice based attack [23, 32]. The key question arises as to how many signatures are needed so as to be able to extract the necessary side channel information to enable the lattice based attack to work. The lattice attack works by constructing a lattice problem from the obtained digital signatures and side channel information, and then applying lattice reduction techniques such as LLL [25] or BKZ [37] to solve the lattice problem. Using this methodology Nguyen and Shparlinski [32], suggest that for an elliptic curve group of order around 160 bits, their probabilistic algorithm would obtain the secret key using an expected 23×2^7 signatures (assuming independent and uniformly at random selected messages) in polynomial time, using only seven consecutive least significant leaked bits of each ephemeral private key. A major issue of their attack in practice is that it seems hard to apply when only a few bits of the underlying ephemeral private key are determined.

Our Contribution: Through the FLUSH+RELOAD attack we are able to obtain a significant proportion of the ephemeral private key bit values, but they are not clustered but in positions spread through the length of the ephemeral private key. As a result, we only obtain for each signature a few (maybe only one) consecutive bits of the ECDSA ephemeral private key, and so the technique described in [32] does not appear at first sight to be instantly applicable. The main contribution of this work is to combine and adapt the FLUSH+RELOAD attack and the lattice techniques. The FLUSH+RELOAD attack is refined to optimise the proportion of information which can be obtained, then the lattice techniques are adapted to utilize the information in the acquired data in an optimal manner. The result is that we are able to reconstruct secret keys for 256 bit elliptic curves with high probability, and low work effort, after obtaining less than 256 signatures.

We illustrate the effectiveness of the attack by applying it to the OpenSSL implementation of ECDSA using a sliding window to compute scalar multiplication, recovering the victims's secret key for the elliptic curve **secp256k1** used in Bitcoin [30]. The implementation of the **secp256k1** curve in OpenSSL is interesting as it uses the wNAF method for exponentiation, as opposed to the GLV method [19], for which the curve was created. It would be an

interesting research topic to see how to apply the FLUSH+RELOAD technique to an implementation which uses the GLV point multiplication method.

In terms of the application to Bitcoin an obvious mitigation against the attack is to limit the number of times a private key is used within the Bitcoin protocol. Each wallet corresponds to a public/private key pair, so this essentially limits the number of times one can spend from a given wallet. Thus, by creating a chain of wallets and transferring Bitcoins from one wallet to the next it is easy to limit the number of signing operations carried out by a single private key. See [9] for a discussion on the distribution of public keys currently used in the Bitcoin network.

The remainder of the paper is organised as follows: In 2 we present the background on ECDSA and the signed sliding window method (or wNAF representation) needed to understand our attack. Then in 3 we present our methodology for applying the FLUSH+RELOAD attack on the OpenSSL implementation of the signed sliding window method of exponentiation. Then in 4 we use the information so obtained to create a lattice problem, and we demonstrate the success probability of our attack.

2 Mathematical Background

In this section we present the mathematical background to our work, by presenting the ECDSA algorithm, and the wNAF/signed window method of point multiplication which is used by OpenSSL to implement ECDSA in the case of curves defined over prime finite fields.

ECDSA: The ElGamal Signature Scheme [20] is the basis of the US 1994 NIST standard, Digital Signature Algorithm (DSA). The ECDSA is the adaptation of one step of this algorithm from the multiplicative group of a finite field to the group of points on an elliptic curve, and is the signature algorithm using elliptic curve cryptography with widescale deployment. In this section we outline the algorithm, so as to fix notation for what follows:

Parameters: The scheme uses as 'domain parameters', which are parameters which can be shared by a large number of users, an elliptic curve E defined over a finite field $\mathbb{F}_q$ and a point $G \in E$ of a large prime order n. The point G is considered as a generator of the group of points of order n. The parameters are chosen as such are generally believed to offer a (symmetric) security level of $\sqrt{n}$ given current knowledge and technologies. The field size q is usually taken to be a large odd prime or a power of 2. The implementation of OpenSSL uses both cases, but in this paper we will focus on the case of q being a large prime.

Public-Private Key pairs: The private key is an integer α, $1 < \alpha < n-1$ and the public key is the point $Q = [\alpha]G$. Calculating the private key from the public key requires solving the ECDLP, which is believed to be hard in practice for correctly chosen parameters. The most efficient currently known algorithms for solving the ECDLP have a square root run time in the size of the group [18, 41], hence the aforementioned security level.

Signing: Suppose Bob, with private-public key pair $\{\alpha, Q\}$, wishes to send a signed message m to Alice. For ECDSA he follows the following steps:

1. Using an approved hash algorithm, compute $e = Hash(m)$, take h to be the integer (modulo n) given by the leftmost ℓ bits of e (where $\ell = \min(\log_2(n)$, the bitlength of the hash)).
2. Randomly select $k \in \mathbb{Z}_n$.
3. Compute the point $(x, y) = [k]G \in E$.
4. Take $r = x \bmod n$; if $r = 0$ then return to step 2.
5. Compute $s = k^{-1}(h + r \cdot \alpha) \bmod n$; if $s = 0$ then return to step 2.
6. Bob sends (m, r, s) to Alice.

Verification: To verify the signature on the message sent by Bob, Alice performs the following steps.

1. Check that all received parameters are correct, that $r, s \in \mathbb{Z}_n$ and that Bob's public key is valid, that is $Q \neq \mathcal{O}$ and $Q \in E$ is of order n.

2. Using the same hash function and method as above, compute $h = Hash(m) \pmod{n}$.
3. Compute $\bar{s} = s^{-1} \bmod n$.
4. Compute the point $(x,y) = [h \cdot \bar{s}]G + [r \cdot \bar{s}]Q$.
5. Verify that $r = x \bmod n$ otherwise reject the signature.

ECDSA is a very brittle algorithm, in the sense that an incorrectly implemented version of Step 2 of the signing algorithm can lead to catastrophic security weaknesses. For example, an inappropriate reuse of the random integer led to the highly publicised breaking of the Sony PS3 implementation of ECDSA. Knowledge of the random value k, often referred to as the *ephemeral key*, leads to knowledge of the secret key, since given a message/signature pair and the corresponding ephemeral key one can recover the secret key via the equation

$$\alpha = (s \cdot k - h) \cdot r^{-1}.$$

It is this equation which we shall exploit in our attack, but we shall do this via obtaining side channel information via a spy process. The spy process targets the computationally expensive part of the signing algorithm, namely Step 3.

Scalar multiplication using wNAF: In OpenSSL Step 3 in the signing algorithm is implemented using the wNAF algorithm. Suppose we wish to compute $[d]P$ for some integer value $d \in [0,\ldots,2^{\ell}]$, the wNAF method utilizes a small amount of pre-processing on P and the fact that addition and subtraction in the elliptic curve group have the same cost, so as to obtain a large performance improvement on the basic binary method of point multiplication. To define wNAF a window size w is first chosen, which for OpenSSL, and the curve **secp256k1**, we have $w = 3$. Then $2^w - 2$ extra points are stored, with a precomputation cost of $2^{w-1} - 1$ point additions, and one point doubling. The values stored are the points $\{\pm G, \pm[3]G, \ldots, \pm[2^w - 1]G\}$.

The next task is to convert the integer d into so called Non-Adjacent From (NAF). This is done by the method in Algorithm 1 which rewrites the integer d as a sum $d = \sum_{i=0}^{\ell-1} d_i \cdot 2^i$, where $d_i \in \{\pm 1, \pm 3, \ldots, \pm(2^w - 1)\}$. The Non-Adjacent From is so named as for any d written in NAF, the output values $d_0, \ldots, d_{\ell-1}$, are such that for every non-zero element d_i there are at least $w+1$ following zero values.

Input: scalar d and window width w
Output: d in wNAF: $d_0, \ldots, d_{\ell-1}$
$\ell \leftarrow 0$
while $d > 0$ **do**
  if d mod 2 = *1* **then**
    $d_\ell \leftarrow d \bmod 2^{w+1}$
    if $d_\ell \geq 2^w$ **then**
      $d_\ell \leftarrow d_\ell - 2^{w+1}$
    end
    $d = d - d_\ell$
  else
    $d_\ell = 0$
  end
  $d = d/2$
  $\ell += 1$
end

Algorithm 1: Conversion to Non-Adjacent Form

Once the integer d has been re-coded into wNAF form, the point multiplication can be carried out by Algorithm 2. The occurrence of a non-zero d_i controls when an addition is performed, with the precise value of d_i determining which point from the list is added.

Before ending this section we note some aspects of the algorithm, and how these are exploited in our attack. A spy process, by monitoring the cache hits/misses, can determine when the code inside the **if–then** block in Algorithm 2 is performed. This happens when the element d_i is non-zero, which reveals the fact that the following $w+1$ values

Input: scalar d in wNAF $d_0, \ldots, d_{\ell-1}$ and precomputed points $\{G, \pm[3]G, \pm[5]G, \ldots, \pm[2^w-1]G\}$
Output: $[d]G$
$Q \leftarrow \mathcal{O}$
for *j from* $\ell - 1$ *downto* 0 **do**
  $Q \leftarrow [2]Q$
  if $d_j \neq 0$ **then**
    $Q \leftarrow Q + [d_j]G$
  end
end

Algorithm 2: Computation of kG using OpenSSL wNAF

$d_{i+1}, \ldots, d_{i+w+1}$ are all zero. This reveals some information about the value d, but not enough to recover the value of d itself.

Instead we focus on the last values of d_i processed by Algorithm 2. We can determine precisely how many least significant bits of d are zero, which means we can determine at least one bit of d, and with probability $1/2$ we determine two bits, with probability $1/4$ we determine three bits and so on. Thus we not only extract information about whether the least significant bits are zero, but we also use the information obtained from the first non-zero bit.

In practice in the OpenSSL code the execution of line 3 is slightly modified. Instead of computing $[k]G$, the code computes $[k + \lambda \cdot n]G$ where $\lambda \in \{1, 2\}$ is chosen such that $\lfloor \log_2(k + \lambda \cdot n) \rfloor = \lfloor \log_2(n) \rfloor + 1$. The fixed size scalar provides protection against the Brumley and Tuveri remote timing attack [11]. For the **secp256k1** curve, n is $2^{256} - \varepsilon$ where $\varepsilon < 2^{129}$. The case $\lambda = 2$, therefore, only occurs for $k < \varepsilon$. As the probability of this case is less than 2^{-125}, we can assume the wNAF algorithm is applied with $d = k + n$.

3 Attacking OpenSSL

In prior work the Montgomery ladder method of point multiplication was shown to be vulnerable to a FLUSH+RELOAD attack [42]. This section discusses the wNAF implementation of OpenSSL and demonstrates that it is also vulnerable. Unlike the side-channel in the Montgomery ladder implementation, which recovers enough bits to allow a direct recovery of the ephemeral private key [42], the side-channel in the wNAF implementation only leaks an average of two bits in each window. Consequently, a further algebraic attack is required to recover the private key. This section describes the FLUSH+RELOAD attack, and its application to the OpenSSL wNAF implementation. The next section completes the recovery of the secret key.

FLUSH+RELOAD is a cache side-channel attack that exploits a property of the Intel implementation of the X86 and X86_64 processor architectures, which allows processes to manipulate the cache of other processes [42, 43].

Using the attack, a spy program can trace or monitor memory read and execute access of a victim program to shared memory pages. The spy program only requires read access to the shared memory pages, hence pages containing binary code in executable files and in shared libraries are susceptible to the attack. Furthermore, pages shared through the use of memory de-duplication in virtualized environments [6, 40] are also susceptible and using them the attack can be applied between co-located virtual machines.

The spy program needs to execute on the same physical processor as the victim, however, unlike most cache-based side channel attacks, our spy monitors access to the last-level cache (LLC). As the LLC is shared between the processing cores of the processor, the spy does not need to execute on the same processing core as the victim. Consequently, the attack is applicable to multi-core processors and is not dependent on hyperthreading or on exploitable scheduler limitations like other published microarchitectural side-channel attacks.

To monitor access to memory, the spy repeatedly evicts the contents of the monitored memory from the LLC, waits for some time and then measures the time to read the contents of the monitored memory. See Algorithm 3 for a pseudo-code of the attack. As reading from the LLC is much faster than reading from memory, the spy can differentiate between these two cases. If, following the wait, the contents of the memory is retrieved from the cache, it indicates that another process has accessed the memory. Thus, by measuring the time to read the contents of the memory, the spy can decide whether the victim has accessed the monitored memory since the last time it was evicted.

```
Input: adrs—the probed address
Output: true if the address was accessed by the victim
begin
    evict(adrs)
    wait_a_bit()
    time ← current_time()
    tmp ← read(adrs)
    readTime ← current_time()-time
    return readTime < threshold
end
```

Algorithm 3: FLUSH+RELOAD Algorithm

Monitoring access to specific memory lines is one of the strengths of the FLUSH+RELOAD technique. Other cache-based tracing techniques monitor access to sets of memory lines that map to the same cache set. The use of specific memory lines reduces the chance of false positives. Capturing the access to the memory line, therefore, indicates that the victim executes and has accessed the line. Consequently, FLUSH+RELOAD does not require any external mechanism to synchronize with the victim.

We tested the attack on an HP Elite 8300 running Fedora 18. The machine features an Intel Core i5-3470 processor, with four execution cores and a 6MB LLC. As the OpenSSL package shipped with Fedora does not support ECC, we used our own build of OpenSSL 1.0.1e. For the experiment we used the curve **secp256k1** which is used by Bitcoin.

For the attack, we used the implementation of FLUSH+RELOAD from [43]. The spy program divides time into time slots of approximately 3,000 cycles (almost 1μs). In each time slot the spy probes memory lines in the group add and double functions. (`ec_GFp_simple_add` and `ec_GFp_simple_dbl`, respectively.) The time slot length is chosen to ensure that there is an empty slot during the execution of each group operation. This allows the spy to correctly distinguish consecutive doubles.

The probes are placed on memory lines which contain calls to the field multiplication function. Memory lines containing call sites are accessed both when the function is called and when it returns. Hence, by probing these memory lines, we reduce the chance of missing accesses due to overlaps with the probes. See [43] for a discussion of overlaps.

To find the memory lines containing the call sites we built OpenSSL with debugging symbols. These symbols are not loaded at run time and do not affect the performance of the code. The debugging symbols are, typically, not available for attackers, however their absence would not present a major obstacle to a determined attacker who could use reverse engineering [16].

4 Lattice Attack Details

We applied the above process on the OpenSSL implementation of ECDSA for the curve **secp256k1**. We fixed a public key $Q = [\alpha]G$, and then monitored via the FLUSH+RELOAD spy process the generation of a set of d signature pairs (r_i, s_i) for $i = 1, \ldots, d$. For each signature pair there is a known hashed message value h_i and an unknown ephemeral private key value k_i.

Using the FLUSH+RELOAD side-channel we also obtained, with very high probability, the sequence of point additions and doubling used when OpenSSL executes the operation $[k_i + n]G$. In particular, this means we learn values c_i and l_i such that

$$k_i + n \equiv c_i \pmod{2^{l_i}},$$

or equivalently

$$k_i \equiv c_i - n \pmod{2^{l_i}}.$$

Where l_i denotes the number of known bits. We can also determine the length of the known run of zeroes in the least significant bits of $k_i + n$, which we will call z_i. In presenting the analysis we assume the d signatures have been selected such that we already know that the value of $k_i + n$ is divisible by 2^Z, for some value of Z, i.e. we pick signatures for

which $z_i \geq Z$. In practice this means that to obtain d such signatures we need to collect (on average) $d \cdot 2^Z$ signatures in total.

We write $a_i = c_i - n \pmod{2^{l_i}}$. For example, writing A for an add, D for a double and X for a *don't know*, we can read off c_i, l_i and z_i from the least execution sequence obtained in the FLUSH+RELOAD analysis. In practice the FLUSH+RELOAD attack is so efficient that we are able to identify A's and D's with almost 100% certainty, with only $\varepsilon = 0.55\% - 0.65\%$ of the symbols turning out to be *don't know*s. To read off the values we use the following table (and its obvious extension), where we present the approximate probability of our attack revealing this sequence.

Sequence	c_i	l_i	z_i	Pr $\approx$
$\ldots X$	0	0.0	0	ε
$\ldots A$	1	1.0	0	$(1-\varepsilon)/2$
$\ldots XD$	0	1.0	1	$\varepsilon \cdot (1-\varepsilon)/2$
$\ldots AD$	2	2.0	1	$((1-\varepsilon)/2)^2$
$\ldots XDD$	0	2.0	2	$\varepsilon \cdot ((1-\varepsilon)/2)^2$
$\ldots ADD$	4	3.0	2	$((1-\varepsilon)/2)^3$

For a given execution of the FLUSH+RELOAD attack, from the table we can determine c_i and l_i, and hence a_i. Then, using the standard analysis from [31, 32], we determine the following values

$$t_i = \lfloor r_i/(2^{l_i} \cdot s_i) \rfloor_n,$$
$$u_i = \lfloor (a_i - h_i/s_i)/2^{l_i} \rfloor_n + n/2^{l_i+1},$$

where $\lfloor \cdot \rfloor_n$ denotes reduction modulo n into the range $[0, \ldots, n)$. We then have that

$$v_i = |\alpha \cdot t_i - u_i|_n < n/2^{l_i+1}, \tag{1}$$

where $|\cdot|_n$ denotes reduction by n, but into the range $(-n/2, \ldots, n/2)$. It is this latter equation which we exploit, via lattice basis reduction, so as to recover d. The key observation found in [31, 32] is that the value v_i is smaller (by a factor of 2^{l_i+1}) than a random integer. Unlike prior work in this area we do not (necessarily) need to just select those executions which give us a "large" value of z_i, say $z_i \geq 3$. Prior work fixes a minimum value of z_i (or essentially equivalently l_i) and utilizes this single value in all equations such as (1). If we do this we would need to throw away all bar $1/2^{z_i+1}$ of the executions obtained. By maintaining full generality, i.e. a variable value of z_i (subject to the constraint $z_i \geq Z$) in each instance of (1), we are able to utilize all information at our disposal and recover the secret key α with very little effort indeed.

The next task is to turn the equations from (1) into a lattice problem. Following [31, 32] we do this in one of two possible ways, which we now recap on.

Attack via CVP: We first consider the lattice $L(B)$ in $d+1$-dimensional real space, generated by the rows of the following matrix

$$B = \begin{pmatrix} 2^{l_1+1} \cdot n & & & \\ & \ddots & & \\ & & 2^{l_d+1} \cdot n & \\ 2^{l_1+1} \cdot t_1 & \ldots & 2^{l_d+1} \cdot t_d & 1 \end{pmatrix}.$$

From (1) we find that there are integers $(\lambda_1, \ldots, \lambda_d)$ such that if we set $\mathbf{x} = (\lambda_1, \ldots, \lambda_d, \alpha)$ and $\mathbf{y} = (2^{l_1+1} \cdot v_1, \ldots, 2^{l_d+1} \cdot v_d, \alpha)$ and $\mathbf{u} = (2^{l_1+1} \cdot u_1, \ldots, 2^{l_d+1} \cdot u_d, 0)$, then we have

$$\mathbf{x} \cdot B - \mathbf{u} = \mathbf{y}.$$

We note that the 2-norm of the vector $\mathbf{y}$ is about $\sqrt{d+1} \cdot n$, whereas the lattice determinant of $L(B)$ is $2^{d+\Sigma l_i} \cdot n^d$. Thus the vector $\mathbf{u}$ is a close vector to the lattice. Solving the Closest Vector Problem (CVP) with input B and $\mathbf{u}$ therefore reveals $\mathbf{x}$ and hence the secret key α.

Attack via SVP: It is often more effective in practice to solve the above CVP problem via the means of embedding the CVP into a Shortest Vector Problem (SVP) in a slightly bigger lattice. In particular we take the lattice $L(B')$ in $d+2$-dimensional real space generated by the rows of the matrix

$$B' = \begin{pmatrix} B & 0 \\ \mathbf{u} & n \end{pmatrix}.$$

This lattice has determinant $2^{d+\Sigma l_i} \cdot n^{(d+1)}$, by taking the lattice vector generated by $\mathbf{x}' = (\mathbf{x}, \alpha, -1)$ we obtain the lattice vector $\mathbf{y}' = \mathbf{x}' \cdot B' = (\mathbf{y}, -n)$. The 2-norm of this lattice vector is roughly $\sqrt{d+2} \cdot n$. We expect the second vector in a reduced basis to be of size $c \cdot n$, and so there is a "good" chance for a suitably strong lattice reduction to obtain a lattice basis whose second vector is equal to $\mathbf{y}'$. Note, the first basis vector is likely to be given by $(-t_1, \ldots, -t_d, n, 0) \cdot B' = (0, \ldots, 0, n, 0)$.

4.1 Experimental Results

To solve the SVP problem we used the BKZ algorithm [37] as implemented in fplll [12]. However, this implementation is only efficient for small block size (say less than 35), due to the fact that BKZ is an exponential algorithm in the block size. Thus for larger block size we implemented a variant of the BKZ-2.0 algorithm [15], however this algorithm is only effective for block sizes β greater than 50. In tuning BKZ-2.0 we used the following strategy, at the end of every round we determined whether we had already solved for the private key, if not we continued, and then gave up after ten rounds. As stated above we applied our attack to the curve **secp256k1**.

We wished to determine what the optimal strategy was in terms of the minimum value of Z we should take, the optimal lattice dimension, and the optimal lattice algorithm. Thus we performed a number of experiments which are reported on in Tables 2, 3 and 4 in Appendix A; where we present our best results obtained for each (d, Z) pair. We also present graphs to show how the different values of β affected the success rate. For each lattice dimension, we measured the optimal parameters as the ones which minimized the value of lattice execution time divided by probability of success. The probability of success was measured by running the attack a number of times, and seeing in how many executions we managed to recover the underlying secret key. We used Time divided by Probability is a crude measure of success, but we note this hides other issues such as expected number of executions of the signature algorithm needed.

All executions were performed on an Intel Xeon CPU running at 2.40 GHz, on a machine with 4GB of RAM. The programs were run in a single thread, and so no advantages where made of the multiple cores on the processor. We ran experiments for the SVP attack using BKZ with block size ranging from 5 to 40 and with BKZ-2.0 with blocksize 50. With our crude measure of Time divided by Probability we find that BKZ with block size 15 or 20 is almost always the method of choice for the SVP method.

We see that the number of signatures needed is consistent with what theory would predict in the case of $Z = 1$ and $Z = 2$, i.e. the lattice reduction algorithm can extract from the side-channel the underlying secret key as soon as the expected number of leaked bits slightly exceeds the number of bits in the secret key. For $Z = 0$ this no longer holds, we conjecture that this is because the lattice algorithms are unable to reduce the basis well enough, in a short enough amount of time, to extract the small amount of information which is revealed by each signature. In other words the input basis for $Z = 0$ is too close to looking like a random basis, unless a large amount of signatures is used.

To solve the CVP problem variant we applied a pre-processing of either fplll or BKZ-2.0. When applying pre-processing of BKZ-2.0 we limited to only one round of execution. We then applied an enumeration technique, akin to the enumeration used in the enumeration sub-routine of BKZ, but centered around the target close vector as opposed to the origin. When a close vector was found this was checked to see whether it revealed the secret key, and if not the enumeration was continued. We restricted the number of nodes in the enumeration tree to 2^{29}, so as to ensure the enumeration did not go on for an excessive amount of time in the cases where the solution vector is hard to find (this mainly affected the experiments in dimension greater than 150). See Tables 5, 6 and 7, in Appendix A, for details of these experiments; again we present the best results for each (d, Z) pair. The enumeration time is highly dependent on whether the close lattice vector is really close to the lattice, thus we see that when the expected number of bits revealed per signature times the number of signatures utilized in the lattice, gets close to the bit size of elliptic curve (256) the

enumeration time drops. Again we see that extensive pre-processing of the basis with more complex lattice reduction techniques provides no real benefit.

The results of the SVP and CVP experiments (Appendix A) show that for fixed Z, increasing the dimension generally decreases the overall expected running time. In some sense, as the dimension increases more information is being added to the lattice and this makes the desired solution vector stand out more. The higher block sizes perform better in the lower dimensions, as the stronger reduction allows them to isolate the solution vector better. The lower block sizes perform better in the higher dimensions, as the high-dimensional lattices already contain much information and strong reduction is not required.

The one exception to this rule is the case of $Z = 2$ in the CVP experiments. In dimensions below 80 the CVP can be solved relatively quickly here, whereas in dimensions 80 up to 100 it takes more time. This can be explained as follows: in the low dimension the CVP-tree is not very big, but contains many solutions. This means that enumeration of the CVP-tree is very quick, but the solution vector is not unique. Thus, the probability of success is equal to the probability of finding the right vector. From dimension 80 upwards, we expect the solution vector to be unique, but the CVP-trees become much bigger on average. If we do not stop the enumeration after a fixed number of nodes, it will find the solution with high probability, but the enumeration takes much longer. Here, the probability of success is the probability of finding a solution at all.

We first note, for both our lattice variants, that there is a wide variation in the probability of success, if we ran a larger batch of tests we would presume this would stabilize. However, even with this caveat we notice a number of remarkable facts. Firstly, recall we are trying to break a 256 bit elliptic curve private key. The conventional wisdom has been that using a window style exponentiation method and a side-channel which only records a distinction between addition and doubling (i.e. does not identify which additions), one would need much more than 256 executions to recover the secret key. However, we see that we have a good chance of recovering the key with less than this. For example, Nguyen and Shparlinksi [32] estimated needing $23 \times 2^7 = 2944$ signatures to recover a 160 bit key, when seven consecutive zero bits of the ephemeral private key were detected. Namely they would use a lattice of dimension 23, but require 2944 signatures to enable to obtain 23 signatures for which they could determine the ones with seven consecutive digits of the ephemeral private key. Note that $23 \cdot 7 = 161 > 160$. Liu and Nguyen [26] extended this attack by using improved lattice algorithms, decreasing the number of signatures required. We are able to have a reasonable chance of success with as little as 200 signatures obtained against a 256 bit key.

In our modification of the lattice attack we not only utilize zero least significant bits, but also notice that the end of a run of zeros tells us that the next bit is one. In addition we utilize all of the run of zeros (say for example eight) and not just some fixed pre-determined number (such as four). This explains our improved lattice analysis, and shows that one can recover the secret with relatively high probability with just a small number of measurements.

As a second note we see that strong lattice reduction, i.e. high block sizes in the BKZ algorithm, or even applying BKZ-2.0, does not seem to gain us very much. Indeed acquiring a few extra samples allows us to drop down to using BKZ with blocksize twenty in almost all cases. Note that in many of our experiments a smaller value of β resulted in a much lower probability of success (often zero), whilst a higher value of β resulted in a significantly increased run time.

Thirdly, we note that if one is unsuccessful on one run, one does not need to derive a whole new set of traces, simply by increasing the number of traces a little bit one can either take a new random sample of the traces one has, or increase the lattice dimension used.

We end by presenting in Table 1 the best variant of the lattice attack, measured in terms of the minimal value of Time divided by Probability of success, for the number of signatures obtained. We see that in a very short amount of time we can recover the secret key from 260 signatures, and with more effort we can even recover it from the FLUSH+RELOAD attack applied to as little at 200 signatures. We see that it is not clear whether the SVP or the CVP approach is the best strategy.

5 Mitigation

As our attack requires capturing multiple signatures, one way of mitigating it is limiting the number of times a private key is used for signing. Bitcoin, which uses the **secp256k1** curve on which this work focuses, recommends using a

Expected # Sigs	SVP/ SVP	d	$Z = \min\{z_i\}$	Pre-Processing and/or SVP Algorithm	Time (s)	Prob Success	$100\times$ Time/Prob
200	SVP	100	1	BKZ ($\beta = 30$)	611.13	3.5	17460
220	SVP	110	1	BKZ ($\beta = 25$)	78.67	2.0	3933
240	CVP	60	2	BKZ ($\beta = 25$)	2.68	0.5	536
260	CVP	65	2	BKZ ($\beta = 10$)	2.26	5.5	41
280	CVP	70	2	BKZ ($\beta = 15$)	4.46	29.5	15
300	CVP	75	2	BKZ ($\beta = 20$)	13.54	53.0	26
320	SVP	80	2	BKZ ($\beta = 20$)	6.67	22.5	29
340	SVP	85	2	BKZ ($\beta = 20$)	9.15	37.0	24
360	SVP	90	2	BKZ ($\beta = 15$)	6.24	23.5	26
380	SVP	95	2	BKZ ($\beta = 15$)	6.82	36.0	19
400	SVP	100	2	BKZ ($\beta = 15$)	7.22	33.5	21
420	SVP	105	2	BKZ ($\beta = 15$)	7.74	43.0	18
440	SVP	110	2	BKZ ($\beta = 15$)	8.16	49.0	16
460	SVP	115	2	BKZ ($\beta = 15$)	8.32	52.0	16
480	CVP	120	2	BKZ ($\beta = 10$)	11.55	87.0	13
500	CVP	125	2	BKZ ($\beta = 10$)	10.74	93.5	12
520	CVP	130	2	BKZ ($\beta = 10$)	10.50	96.0	11
540	SVP	135	2	BKZ ($\beta = 10$)	7.44	55.0	13

Table 1. Combined Results. The best lattice parameter choice for each number of signatures obtained (in steps of 20)

new key for each transaction [30]. This recommendation, however, is not always followed [36], exposing users to the attack.

Another option to reduce the effectiveness of the FLUSH+RELOAD part of the attack would be to exploit the inherent properties of this "Koblitz" curve within the OpenSSL implementation; which would also have the positive side result of speeding up the scalar multiplication operation. The use of the *GLV method* [19] for point multiplication would not completely thwart the above attack, but, in theory, reduces its effectiveness. The GLV method is used to speed up the computation of point scalar multiplication when the elliptic curve has an efficiently computable endomorphism. This partial solution is only applicable to elliptic curves with easily computable automorphisms with sufficiently large automorphism group; such as the curve **secp256k1** which we used in our example.

The curve **secp256k1** is defined over a prime field of characteristic p with $p \equiv 1 \mod 6$. This means that $\mathbb{F}_p$ contains a primitive 6th root of unity ζ and if (x,y) is in the group of points on E, then $(-\zeta x, y)$ is also. In fact, $(-\zeta x, y) = [\lambda](x,y)$ for some $\lambda^6 = 1 \mod n$. Since the computation of $(-\zeta x, y)$ from (x,y) costs only one finite field multiplication (far less than computing $[\lambda](x,y)$) this can be used to speed up scalar multiplication: instead of computing $[k]G$, one computes $[k_0]G + [k_1]([\lambda]G)$ where k_0, k_1 are around the size of $k^{1/2}$. This is known to be one of the fastest methods of performing scalar multiplication [19]. The computation of $[k_0]G + [k_1]([\lambda]G)$ is not done using two scalar multiplications then a point addition, but uses the so called *Straus-Shamir* trick which used joint double and add operations [19, Alg 1] performing the two scalar multiplications and the addition simultaneously.

The GLV method alone would be vulnerable to simple side-channel analysis. It is necessary to re-code the scalars k_0 and k_1 and comb method as developed and assembled in [17] so that the execution is regular to thwart simple power analysis and timing attacks. Using the attack presented above we are able to recover around 2 bits of the secret key for each signature monitored. If the GLV method were used in conjunction with wNAF, the number of bits (on average) leaked per signature would be reduced to $4/3$. It is also possible to extend the GLV method to representations of k in terms of higher degrees of λ, for example writing $k = k_0 + k_1\lambda + \cdots + k_t\lambda^t \mod n$. For $t = 2$ the estimated rate of bit leakage would be $6/7$ bits per signature (though this extension is not possible for the example curve due to the order of the automorphism).

We see that using the GLV method can reduce the number of leaked bits but it is not sufficient to prevent the attack. A positive flip side of this and the attack of [42] is that implementing algorithms which will improve the efficiency of the scalar multiplication seem, at present, to reduce the effectiveness of the attacks.

Scalar blinding techniques [10, 27] use arithmetic operations on the scalar to hide the value of the scalar from potential attackers. The method suggested by these works is to compute $[(k+m\cdots n+\bar{m})]G-[\bar{m}]G$ where m and $\bar{m}$ are small (e.g. 32 bits) numbers. The random values used mask the bits of the scalar and prevent the spy from recovering the scalar from the leaked data.

The information leak in our attack originates from using the sliding window in the wNAF algorithm for scalar multiplication. Hence, an immediate fix for the problem is to use a fixed window algorithm for scalar multiplication. A naïve implementation of a fixed window algorithm may still be vulnerable to the PRIME+PROBE attack, e.g. by adapting the technique of [35]. To provide protection against the attack, the implementation must prevent any data flow from sensitive key data to memory access patterns. Methods for achieving this are used in NaCL [8], which ensures that the sequence of memory accesses it performs is not dependent on the private key. A similar solution is available in the implementation of modular exponentiation in OpenSSL, where the implementation attempts to access the same sequence of memory lines irrespective of the private key. However, this approach may leak information [7, 39].

Acknowledgements

The first and fourth authors wish to thank Dr Katrina Falkner for her advice and support and the Defence Science and Technology Organisation (DSTO) Maritime Division, Australia, who partially funded their work. The second and third authors work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant EP/I03126X, and by Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079[3].

References

1. Onur Acıiçmez. Yet another microarchitectural attack: exploiting I-Cache. In Peng Ning and Vijay Atluri, editors, *Proceedings of the ACM Workshop on Computer Security Architecture*, pages 11–18, Fairfax, Virginia, United States, November 2007.
2. Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In Stefan Mangard and François-Xavier Standaert, editors, *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, pages 110–124, Santa Barbara, California, United States, August 2010.
3. Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In Steven D. Galbraith, editor, *Proceedings of the 11th IMA International Conference on Cryptography and Coding*, volume 4887 of *Lecture Notes in Computer Science*, pages 185–203, Cirencester, United Kingdom, December 2007.
4. Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the Second ACM Symposium on Information, Computer and Communication Security*, pages 312–320, Singapore, March 2007.
5. Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In Tal Malkin, editor, *Proceedings of the Cryptographers' Track at the RSA Conference*, pages 256–273, San Francisco, California, United States, April 2008.
6. Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium*, pages 19–28, Montreal, Quebec, Canada, July 2009.
7. Daniel J. Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, April 2005.
8. Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Proceedings of the 2nd international conference on Cryptology and Information Security in Latin America*, LATINCRYPT'12, pages 159–176, Berlin, Heidelberg, 2012. Springer-Verlag.
9. Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. Cryptology ePrint Archive, Report 2013/734, 2013. `http://eprint.iacr.org/`.
10. Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 667–684. Springer-Verlag, 2009.

[3] The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

11. Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Diaz, editors, *Computer Security - ESORICS 2011*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer-Verlag, 2011.
12. David Cadé, Xavier Pujol, and Damien Stehlé. Fplll-4.0.4. `http://perso.ens-lyon.fr/damien.stehle/fplll/`, 2013.
13. Anne Canteaut, Cédric Lauradoux, and André Seznec. Understanding cache attacks. Technical Report 5881, INRIA, April 2006.
14. CaiSen Chen, Tao Wang, YingZhan Kou, XiaoCen Chen, and Xiong Li. Improvement of trace-driven I-Cache timing attack on the RSA algorithm. *The Journal of Systems and Software*, 86(1):100–107, 2013.
15. Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *Advances in Cryptology - ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.
16. Teodoro Cipresso and Mark Stamp. Software reverse engineering. In Peter Stavroulakis and Mark Stamp, editors, *Handbook of Information and Communication Security*, chapter 31, pages 659–696. Springer, 2010.
17. Armando Faz-Hernandez, Patrick Longa, and Ana H. Sanchez. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. Cryptology ePrint Archive, Report 2013/158, 2013. `http://eprint.iacr.org/`.
18. Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Improving the parallelized pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000.
19. Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 2001.
20. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
21. K. Gopalakrishnan, Nicolas Thériault, and Chui Zhi Yao. Solving discrete logarithms from partial knowledge of the key. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *Progress in Cryptology – INDOCRYPT 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages 224–237. Springer, 2007.
22. David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games — bringing access-based cache attacks on AES to practice. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 490–595, Oakland, California, United States, May 2011.
23. Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, 2001.
24. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
25. A.K. Lenstra, H.W.jun. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515 –534, 1982.
26. Mingjie Liu and Phong Q. Nguyen. Solving BDD by enumeration: An update. In Ed Dawson, editor, *CT-RSA*, volume 7779 of *Lecture Notes in Computer Science*, pages 293–309. Springer, 2013.
27. Bodo Möller. Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks. In Agnes Hui Chan and Virgil D. Gligor, editors, *Proceedings of the fifth International Conference on Information Security*, number 2433 in Lecture Notes in Computer Science, pages 402–413, São Paulo, Brazil, September 2002.
28. James A. Muir and Douglas R. Stinson. On the low Hamming weight discrete logarithm problem for nonadjacent representations. *Appl. Algebra Eng. Commun. Comput.*, 16(6):461–472, 2006.
29. David Naccache, Phong Q. Nguên, Michael Tunstall, and Claire Whealan. Experimenting with faults, lattices and the DSA. In Serge Vaudenay, editor, *Public Key Cryptography*, volume 3386 of *Lecture Notes in Computer Science*, pages 16–28, Les Diablerets, Switzerland, January 2005. Springer.
30. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`.
31. Phong Q. Nguyen and Igor Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *J. Cryptology*, 15(3):151–176, 2002.
32. Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, Codes and Cryptography*, 30(2):201–217, September 2003.
33. OpenSSL. `http://www.openssl.org`.
34. Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169, 2002.
35. Colin Percival. Cache missing for fun and profit. `http://www.daemonology.net/papers/htt.pdf`, 2005.
36. Dorit Ron and Adi Shamir. Quantitative analysis of the full Bitcoin transaction graph. Cryptology ePrint Archive, Report 2012/584, 2012. `http://eprint.iacr.org/`.
37. Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In *Fundamentals of Computation Theory – FCT 1991*, volume 529 of *Lecture Notes in Computer Science*, pages 68–85. Springer, 1991.

38. Douglas R. Stinson. Some baby-step giant-step algorithms for the low Hamming weight discrete logarithm problem. *Math. Comput.*, 71(237):379–391, 2002.
39. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks in AES, and countermeasures. *Journal of Cryptology*, 23(2):37–71, January 2010.
40. Carl A. Waldspurger. Memory resource management in VMware ESX Server. In David E. Culler and Peter Druschel, editors, *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 181–194, Boston, Massachusetts, United States, December 2002.
41. Michael J. Wiener and Robert J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In Stafford E. Tavares and Henk Meijer, editors, *Selected Areas in Cryptography*, volume 1556 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 1998.
42. Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014. `http://eprint.iacr.org/`.
43. Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, 2014. To appear.
44. Yinqian Zhang, Ari Jules, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *Proceedings of the 19th ACM Conference on Computer and Communication Security*, pages 305–316, Raleigh, North Carolina, United States, October 2012.

A Experimental Results

d	Algorithm	Expected # Sigs	Lattice Time (s)	Prob.0 Success	$100\times$ Time/Prob
240	BKZ ($\beta = 25$)	240	212.01	8.0	2125
245	BKZ ($\beta = 20$)	245	50.78	2.5	2031
250	BKZ ($\beta = 20$)	250	52.08	2.5	2083
255	BKZ ($\beta = 20$)	255	53.60	3.0	1786
260	BKZ ($\beta = 20$)	260	52.93	6.0	882
265	BKZ ($\beta = 20$)	265	54.97	8.5	646
270	BKZ ($\beta = 15$)	270	35.48	3.5	1013
275	BKZ ($\beta = 20$)	275	55.30	12.5	442
280	BKZ ($\beta = 20$)	280	58.55	11.5	508
285	BKZ ($\beta = 20$)	285	61.56	16.0	384
290	BKZ ($\beta = 20$)	290	67.47	18.5	364
295	BKZ ($\beta = 15$)	295	43.92	9.5	462
300	BKZ ($\beta = 20$)	300	73.30	20.0	366
305	BKZ ($\beta = 20$)	305	78.09	27.0	289
310	BKZ ($\beta = 20$)	310	83.01	29.0	286
315	BKZ ($\beta = 20$)	315	87.70	29.0	302
320	BKZ ($\beta = 20$)	320	93.28	30.0	310
325	BKZ ($\beta = 20$)	325	91.54	22.0	416
330	BKZ ($\beta = 15$)	330	63.34	21.0	301
335	BKZ ($\beta = 15$)	335	64.28	23.0	279

Table 2. SVP Analysis Experimental Results : $Z = \min z_i = 0$

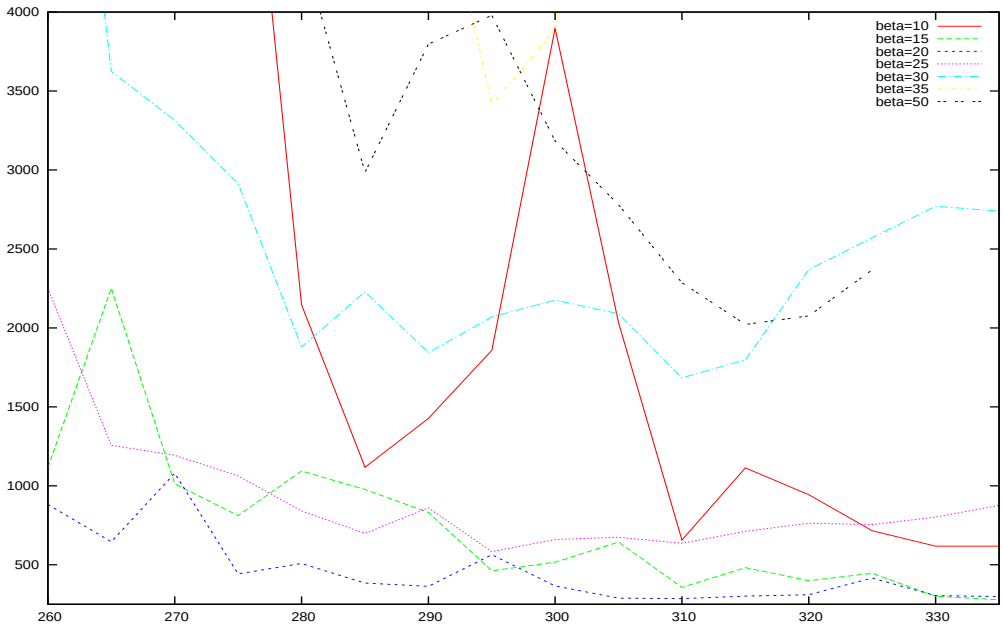

Fig. 1. SVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 0$

d	Algorithm	Expected # Sigs	Lattice Time (s)	Prob.0 Success	$100\times$ Time/Prob
100	BKZ ($\beta = 30$)	200	611.13	3.5	17460
105	BKZ ($\beta = 30$)	210	702.67	7.5	9368
110	BKZ ($\beta = 25$)	220	78.67	2.0	3933
115	BKZ ($\beta = 25$)	230	71.18	3.5	2033
120	BKZ ($\beta = 20$)	240	14.78	1.0	1478
125	BKZ ($\beta = 10$)	250	6.81	1.0	681
130	BKZ ($\beta = 20$)	260	15.12	4.0	378
135	BKZ ($\beta = 25$)	270	57.83	20.0	289
140	BKZ ($\beta = 20$)	280	16.47	9.0	182
145	BKZ ($\beta = 25$)	290	57.63	29.5	195
150	BKZ ($\beta = 20$)	300	19.05	17.0	112
155	BKZ ($\beta = 15$)	310	13.14	13.5	97
160	BKZ ($\beta = 15$)	320	14.00	16.0	87
165	BKZ ($\beta = 15$)	330	15.75	17.5	90
170	BKZ ($\beta = 15$)	340	17.09	23.0	74
175	BKZ ($\beta = 15$)	350	18.14	23.0	78

Table 3. SVP Analysis Experimental Results : $Z = \min z_i = 1$

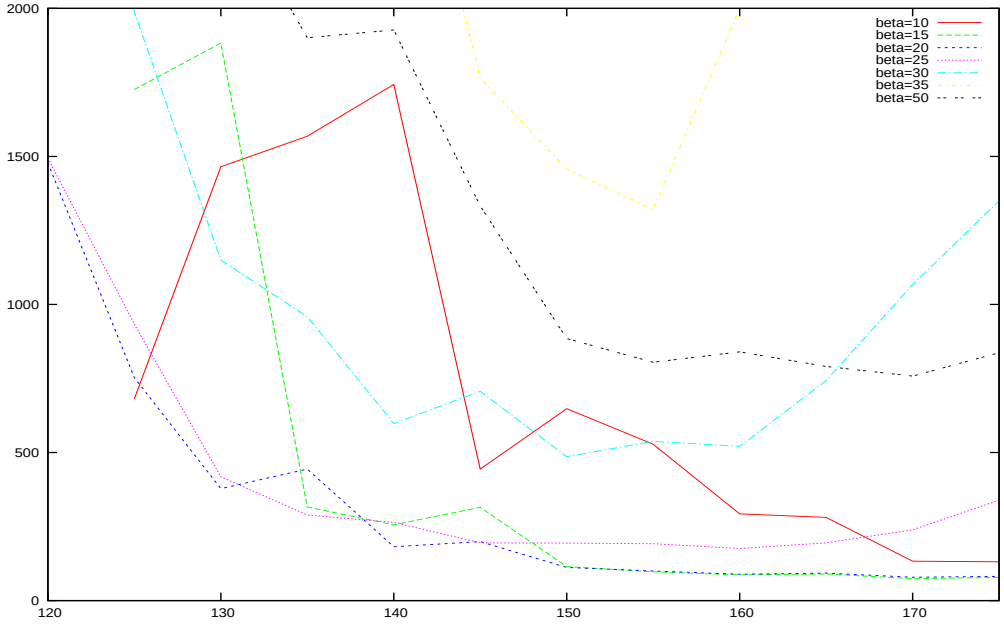

Fig. 2. SVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 1$

d	Algorithm	Expected # Sigs	Lattice Time (s)	Prob.0 Success	$100\times$ Time/Prob
65	BKZ ($\beta = 25$)	260	5.17	2.5	206
70	BKZ ($\beta = 25$)	280	7.93	13.5	58
75	BKZ ($\beta = 25$)	300	13.58	23.5	57
80	BKZ ($\beta = 20$)	320	6.67	22.5	29
85	BKZ ($\beta = 20$)	340	9.15	37.0	24
90	BKZ ($\beta = 15$)	360	6.24	23.5	26
95	BKZ ($\beta = 15$)	380	6.82	36.0	19
100	BKZ ($\beta = 15$)	400	7.22	33.5	21
105	BKZ ($\beta = 15$)	420	7.74	43.0	18
110	BKZ ($\beta = 15$)	440	8.16	49.0	16
115	BKZ ($\beta = 15$)	460	8.32	52.0	16
120	BKZ ($\beta = 10$)	480	6.49	44.0	14
125	BKZ ($\beta = 10$)	500	6.83	45.0	14
130	BKZ ($\beta = 10$)	520	7.06	48.0	14
135	BKZ ($\beta = 10$)	540	7.44	55.0	13

Table 4. SVP Analysis Experimental Results : $Z = \min z_i = 2$

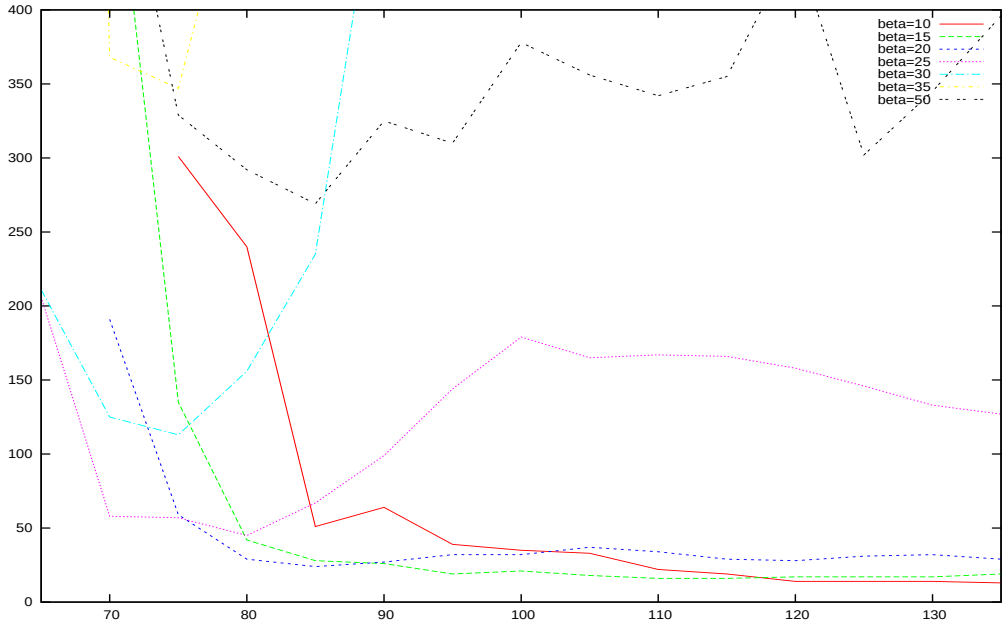

Fig. 3. SVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 2$

d	Pre-Processing Algorithm	Expected # Sigs	Time (s)	Prob.0 Success	$100\times$ Time/Prob
300	BKZ ($\beta = 10$)	300	100.09	5.0	2002
305	BKZ ($\beta = 20$)	305	186.61	13.5	1382
310	BKZ ($\beta = 10$)	310	110.12	7.0	1573
315	BKZ ($\beta = 10$)	315	114.22	10.0	1142
320	BKZ ($\beta = 10$)	320	125.69	10.5	1197
325	BKZ ($\beta = 20$)	325	246.89	22.5	1097
330	BKZ ($\beta = 15$)	330	153.59	16.0	960
335	BKZ ($\beta = 15$)	335	162.22	24.5	662
340	BKZ ($\beta = 15$)	340	167.08	19.0	879
345	BKZ ($\beta = 15$)	345	178.54	30.0	595
350	BKZ ($\beta = 15$)	350	191.91	30.5	629
355	BKZ ($\beta = 15$)	355	194.37	32.0	607
360	BKZ ($\beta = 15$)	360	198.39	34.0	583
365	BKZ ($\beta = 15$)	365	216.43	44.5	486
370	BKZ ($\beta = 15$)	370	218.68	44.5	491
375	BKZ ($\beta = 15$)	375	228.25	45.5	502
380	BKZ ($\beta = 10$)	380	187.14	40.0	468
385	BKZ ($\beta = 15$)	385	243.71	58.0	420
390	BKZ ($\beta = 15$)	390	249.26	61.0	409
395	BKZ ($\beta = 10$)	395	213.76	43.5	491

Table 5. CVP Analysis Experimental Results : $Z = \min z_i = 0$

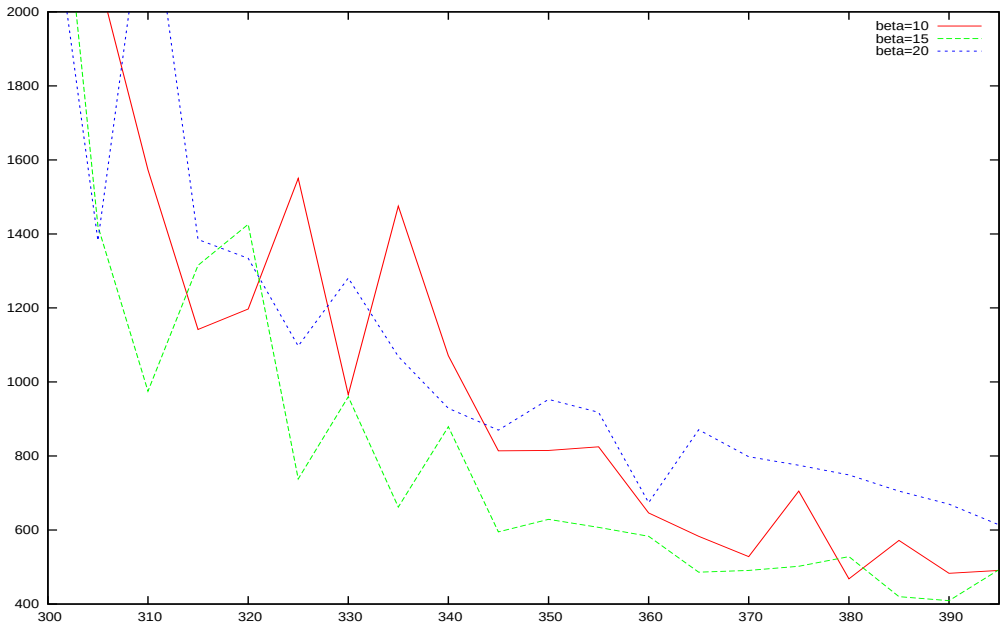

Fig. 4. CVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 0$

d	Pre-Processing Algorithm	Expected # Sigs	Time (s)	Prob.0 Success	$100\times$ Time/Prob
150	BKZ ($\beta = 15$)	300	32.43	3.0	1081
155	BKZ ($\beta = 15$)	310	33.90	8.0	424
160	BKZ ($\beta = 20$)	320	48.26	13.5	357
165	BKZ ($\beta = 20$)	330	50.97	20.0	255
170	BKZ ($\beta = 15$)	340	39.58	22.0	180
175	BKZ ($\beta = 15$)	350	41.20	26.0	158
180	BKZ ($\beta = 15$)	360	43.50	31.5	138
185	BKZ ($\beta = 15$)	370	44.30	39.5	112
190	BKZ ($\beta = 15$)	380	45.98	42.0	109
195	BKZ ($\beta = 15$)	390	46.15	46.0	100
200	BKZ ($\beta = 15$)	400	45.41	60.5	75
205	BKZ ($\beta = 15$)	410	48.45	65.5	74
210	BKZ ($\beta = 10$)	420	41.89	59.5	70
215	BKZ ($\beta = 15$)	430	49.56	76.0	65
220	BKZ ($\beta = 15$)	440	49.88	86.0	58
225	BKZ ($\beta = 10$)	450	44.58	77.0	58
230	BKZ ($\beta = 15$)	460	53.23	92.0	58
235	BKZ ($\beta = 10$)	470	52.86	88.0	60
240	BKZ ($\beta = 10$)	480	48.37	90.5	53
245	BKZ ($\beta = 10$)	490	49.74	89.5	56

Table 6. CVP Analysis Experimental Results : $Z = \min z_i = 1$

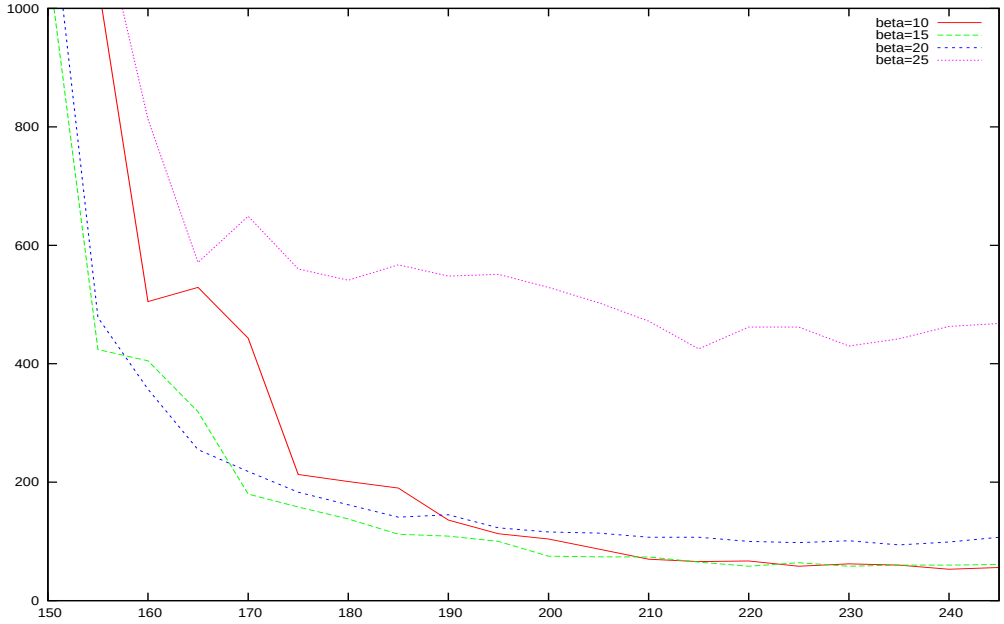

Fig. 5. CVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 1$

d	Pre-Processing Algorithm	Expected # Sigs	Time (s)	Prob.0 Success	$100\times$ Time/Prob
60	BKZ ($\beta = 25$)	240	2.68	0.5	536
65	BKZ ($\beta = 10$)	260	2.26	5.5	41
70	BKZ ($\beta = 15$)	280	4.46	29.5	15
75	BKZ ($\beta = 20$)	300	13.54	53.0	26
80	BKZ ($\beta = 20$)	320	21.83	17.0	128
85	BKZ ($\beta = 15$)	340	20.08	25.5	130
90	BKZ ($\beta = 20$)	360	23.36	35.0	67
95	BKZ ($\beta = 20$)	380	22.40	52.5	43
100	BKZ ($\beta = 20$)	400	22.95	67.0	34
105	BKZ ($\beta = 20$)	420	21.76	77.0	28
110	BKZ ($\beta = 15$)	440	14.74	81.0	18
115	BKZ ($\beta = 15$)	460	14.82	86.5	17
120	BKZ ($\beta = 10$)	480	11.55	87.0	13
125	BKZ ($\beta = 10$)	500	10.74	93.5	12
130	BKZ ($\beta = 10$)	520	10.50	96.0	11

Table 7. CVP Analysis Experimental Results : $Z = \min z_i = 2$

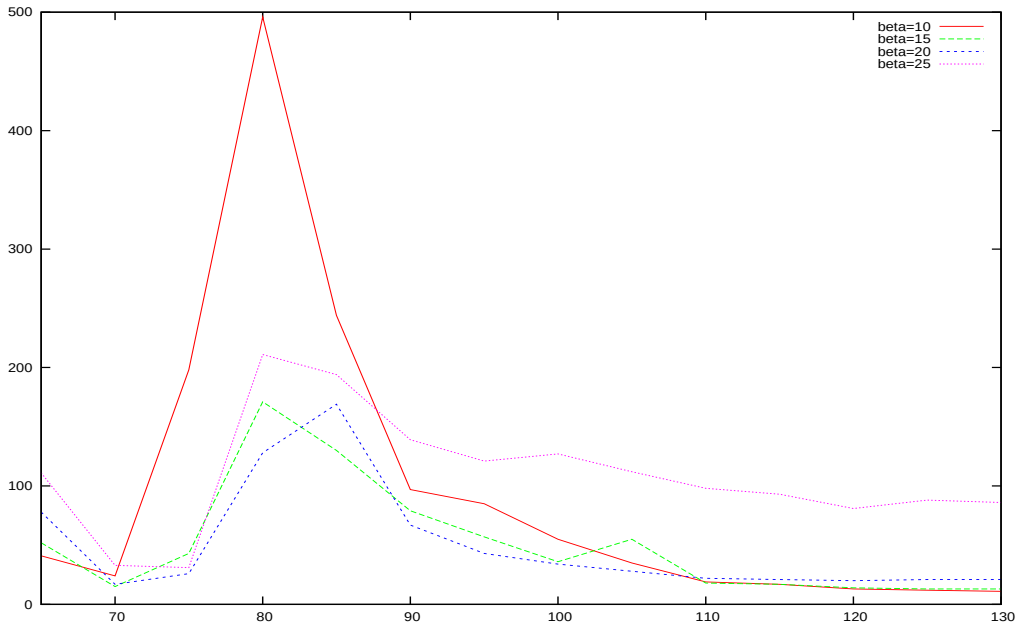

Fig. 6. CVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 2$

Just A Little Bit More

Joop van de Pol[1], Nigel P. Smart[1], and Yuval Yarom[2]

[1] Dept. Computer Science, University of Bristol, United Kingdom.
joop.vandepol@bristol.ac.uk,nigel@cs.bris.ac.uk
[2] School of Computer Science, The University of Adelaide, Australia.
yval@cs.adelaide.edu.au

Abstract. We extend the FLUSH+RELOAD side-channel attack of Benger et al. to extract a significantly larger number of bits of information per observed signature when using OpenSSL. This means that by observing only 25 signatures, we can recover secret keys of the **secp256k1** curve, used in the Bitcoin protocol, with a probability greater than 50 percent. This is an order of magnitude improvement over the previously best known result.
The new method of attack exploits two points: Unlike previous partial disclosure attacks we utilize all information obtained and not just that in the least significant or most significant bits, this is enabled by a property of the "standard" curves choice of group order which enables extra bits of information to be extracted. Furthermore, whereas previous works require direct information on ephemeral key bits, our attack utilizes the indirect information from the wNAF double and add chain.

1 Introduction

The Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve analogue of the Digital Signature Algorithm (DSA). It has been well known for over a decade that the randomization used within the DSA/ECDSA algorithm makes it susceptible to side-channel attacks. In particular a small leakage of information on the ephemeral secret key utilized in each signature can be combined over a number of signatures to obtain the entire key.

Howgrave-Graham and Smart [14] showed that DSA is vulnerable to such partial ephemeral key exposure and their work was made rigorous by Nguyen and Shparlinski [21], who also extended these results to ECDSA [22]. More specifically, if, for a polynomially bounded number of random messages and ephemeral keys about $\log^{1/2} q$ least significant bits (LSBs) are known, the secret key α can be recovered in polynomial time. A similar result holds for a consecutive sequence of the most significant bits (MSBs), with a potential need for an additional leaked bit due to the paucity of information encoded in the most significant bit of the ephemeral key. When an arbitrary sequence of consecutive bits in the ephemeral key is known, about twice as many bits are required. The attack works by constructing a lattice problem from the obtained digital signatures and side-channel information, and then applying lattice reduction techniques such as LLL [16] or BKZ [23] to solve said lattice problem.

Brumley and co-workers employ this lattice attack to recover ECDSA keys using leaked LSBs (in [4]) and leaked MSBs (in [5]). The former uses a cache side-channel to extract the leaked information and the latter exploits a timing side-channel. In both attacks, a fixed number of bits from each signature is used and signatures are used only if the values of these bits are all zero. Signatures in which the value of any of these bits are one are ignored. Consequently, both attacks require more than 2,500 signatures to break a 160-bit private key.

More recently, again using a cache based side-channel, Benger et al. [2] use the LSBs of the ephemeral key for a wNAF (a.k.a. sliding window algorithm) multiplication technique. By combining a new side-channel called the FLUSH+RELOAD side-channel [26, 27], and a more precise lattice attack strategy, which utilizes all of the leaked LSBs from every signature, Benger et al. are able to significantly reduces the number of signatures required. In particular they report that the full secret key of a 256-bit system can be recovered with about 200 signatures in a reasonable length of time, and with a reasonable probability of success.

In this work we extend the FLUSH+RELOAD technique of Benger et al. to reduce the number of required signatures by an order of magnitude. Our methodology abandons the concentration on extraction of bits in just the MSB and LSB positions, and instead focuses on all the information leaked by all the bits of the ephemeral key. In particular we exploit a property of many of the standardized elliptic curves as used in OpenSSL. Our method, just as in [2], applies the FLUSH+RELOAD side-channel technique to the wNAF elliptic curve point multiplication algorithm in OpenSSL.

ECDSA Using Standard Elliptic Curves: The domain parameters for ECDSA are an elliptic curve E over a field $\mathbb{F}$, and a point G on E, of order q. Given a hash function h, the ECDSA signature of a message m, with a private key $0 < \alpha < q$ and public key $Q = \alpha G$, is computed by:

- Selecting a random ephemeral key $0 < k < q$
- Computing $r = x(kG) \pmod{q}$, the X coordinate of kG.
- Computing $s = k^{-1}(h(m) + \alpha \cdot r) \pmod{q}$.

The process is repeated if either $r = 0$ or $s = 0$. The pair (r, s) is the signature.

To increase interoperability, standard bodies have published several sets of domain parameters for ECDSA [1, 7, 20]. The choice of moduli for the fields used in these standard curves is partly motivated by efficiency arguments. For example, all of the moduli in the curves recommended by FIPS [20] are generalised Mersenne primes [24] and many of them are pseudo-Mersenne primes [10]. This choice of moduli facilitates efficient modular arithmetic by avoiding a division operation which may otherwise be required.

A consequence of using pseudo-Mersenne primes as moduli is that, due to Hasse's Theorem, not only is the finite-field order close to a power of two, but so is the elliptic-curve group order.

That is, q can be expressed as $2^n - \varepsilon$, where $|\varepsilon| < 2^p$ for some $p \approx n/2$. We demonstrate that such curves are more susceptible to partial disclosure of ephemeral keys than was hitherto known. This property increases the amount of information that can be used from partial disclosure and allows for a more effective attack on ECDSA.

Our Contribution: We demonstrate that the above property of the standardized curves allows the utilization of far more leaked information, in particular some arbitrary sequences of consecutive leaked bits. In a nutshell, adding or subtracting q to or from an unknown number is unlikely to change any bits in positions between $p+1$ and n. Based on this observation we are able to use (for wNAF multiplication algorithms) all the information in consecutive bit sequences in positions above $p+1$. Since in many of the standard curves $p \approx n/2$, a large amount of information is leaked per signature. (Assuming one can extract the sequence of additions and doubles in an algorithm.) As identified by Ciet and Joye [8] and exploited by Feix et al. [11], the same property also implies that techniques for mitigating side-channel attack, such as the scalar blinding suggested in [4, 18], do not protect bits in positions above $p+1$.

Prior works deal with the case of partial disclosure of consecutive sequences of bits of the ephemeral key. Our work offers two improvements: It demonstrates how to use partial information leaked from the double and add chains of the wNAF scalar multiplication algorithm [13, 19]. In most cases, the double and add chain does not provide direct information on the value of bits. It only identifies sequences of repeating bits without identifying the value of these bits. We show how to use this information to construct a lattice attack on the private key. Secondly, our attack does not depend on the leaked bits being consecutive. We use information leaked through the double and add chain even though it is spread out along the ephemeral key.

By using more leaked information and exploiting the above property of the elliptic curves, our attack only requires a handful of leaked signatures to fully break the private key. Our experiments show that the perfect information leaked on double and add chains of only 13 signatures is sufficient for recovering the 256 bit private key of the **secp256k1** curve with probability greater than 50 percent. For the 521 bit curve **secp521r1**, 40 signatures are required. We further demonstrate that for the **secp256k1** case observing 25 signatures is highly likely to recover 13 perfect double and add chains. Hence, by observing 25 Bitcoin transactions using the same key, an attacker can expect to recover the private key. For most of the paper we discuss the case of perfect side channels which result in perfect double and add chains, then in Section 6 we show how this assumption can be removed in the context of a real FLUSH+RELOAD attack.

2 Background

In this section we discuss three basic procedures we will be referring to throughout. Namely the FLUSH+RELOAD side-channel attack technique, wNAF scalar multiplication method and the use of lattices to extract secret keys from triples. The side-channel information we obtain from executing the wNAF algorithm produces instances of the Hidden Number Problem (HNP) [3]. Since the HNP is traditionally studied via lattice reduction it is therefore not surprising that we are led to lattice reduction in our analysis.

2.1 The FLUSH+RELOAD Side-Channel Attack Technique

FLUSH+RELOAD is a recently discovered cache side-channel attack [26, 27]. The attack exploits a weakness in the Intel implementation of the popular X86 architecture, which allows a spy program to monitor other programs' read or execute access to shared regions of memory. The spy program only requires read access to the monitored memory.

Unlike most cache side-channel attacks, FLUSH+RELOAD uses the Last-Level Cache (LLC), which is the cache level closest to the memory. The LLC is shared by the execution cores in the processor, allowing the attack to operate when the spy and victim processes execute on different cores. Furthermore, as most virtual machine hypervisors (VMMs) actively share memory between co-resident virtual machines, the attack is applicable to virtualized environment and works cross-VM.

```
Input: adrs—the probed address
Output: true if the address was accessed by the victim
begin
    evict(adrs)
    wait_a_bit()
    time ← current_time()
    tmp ← read(adrs)
    readTime ← current_time()-time
    return readTime < threshold
end
```

Algorithm 1: FLUSH+RELOAD Algorithm

To monitor access to memory, the spy repeatedly evicts the contents of the monitored memory from the LLC, waits for some time and then measures the time to read the contents of the monitored memory. See Algorithm 1 for a pseudo-code of the attack. FLUSH+RELOAD uses the X86 `clflush` instruction to evict contents from the cache. To measure time the spy uses the `rdtsc` instruction which returns the time since processor reset measured in processor cycles.

As reading from the LLC is much faster than reading from memory, the spy can differentiate between these two cases. If, following the wait, the contents of memory is retrieved from the cache, it indicates that another process has accessed the memory. Thus, by measuring the time to read the contents of memory, the spy can decide whether the victim has accessed the monitored memory since the last time it was evicted.

To implement the attack, the spy needs to share the monitored memory with the victim. For attacks occurring within the same machine, the spy can map files used by the victim into its own address space. Examples of these files include the victim program file, shared libraries or data files that the victim accesses. As all mapped copies of files are shared, this gives the spy access to memory pages accessed by the victim. In virtualized environments, the spy does not have access to the victim's files. The spy can, however, map copies of the victim files to its own address space, and rely on the VMM to merge the two copies using page de-duplication [15, 25]. It should be pointed that, as the LLC is physically tagged, the virtual address in which the spy maps the files is irrelevant for the attack. Hence, FLUSH+RELOAD is oblivious to address space layout randomization [17].

This sharing only works when the victim does not make private modifications to the contents of the shared pages. Consequently, many FLUSH+RELOAD attacks target executable code pages, monitoring the times the victim executes specific code. The spy typically divides time into fixed width time slots. In each time slot the spy monitors a few memory locations and records the times that these locations were accessed by the victim. By reconstructing a trace of victim access, the spy is able to infer the data the victim is operating on. Prior works used this attack to recover the private key of GnuPG RSA [27] as well as for recovering the ephemeral key used in OpenSSL ECDSA signatures either completely, for curves over binary fields [26], or partially, for curves over prime fields [2].

2.2 The wNAF Scalar Multiplication Method

Several algorithms for computing the scalar multiplication kG have been proposed. One of the suggested methods is to use the *windowed nonadjacent form* (wNAF) representation of the scalar k, see [13]. In wNAF a number is represented by a sequence of digits k_i. The value of a digit k_i is either zero or an odd number $-2^w < k_i < 2^w$, with each pair of

non-zero digits separated by at least w zero digits. The value of k can be calculated from its wNAF representation using $k = \sum 2^i \cdot k_i$. See Algorithm 2 for a method to convert a scalar k into its wNAF representation. We use $|\cdot|_x$ to denote the reduction modulo x into the range $[-x/2, \ldots, x/2)$.

```
Input: Scalar k and window width w
Output: k in wNAF: k_0, k_1, k_2 ...
begin
    e ← k
    i ← 0
    while e > 0 do
        if e mod 2 = 1 then
            k_i ← |e|_{2^{w+1}}
            e ← e − k_i
        else
            k_i ← 0
        end
        e ← e/2
        i ← i + 1
    end
end
```

Algorithm 2: Conversion to Non-Adjacent Form

Let $\overline{k_i}$ be the value of the variable e at the start of the i^{th} iteration in Algorithm 2. From the algorithm, it is clear that

$$k_i = \begin{cases} 0 & \overline{k_i} \text{ is even} \\ |\overline{k_i}|_{2^{w+1}} & \overline{k_i} \text{ is odd} \end{cases} \tag{1}$$

Furthermore:

$$k = 2^i \cdot \overline{k_i} + \sum_{j<i} 2^j \cdot k_j \tag{2}$$

Let m and $m+l$ be the position of two consecutive non-zero wNAF digits, i.e. $k_m, k_{m+l} \neq 0$ and $k_{m+i} = 0$ for all $0 < i < l$. We now have

$$-2^{m+w} < \sum_{i \leq m} k_i \cdot 2^i < 2^{m+w}, \tag{3}$$

and because $l > w$, we get $-2^{m+l-1} < \sum_{i \leq m+l-1} k_i \cdot 2^i < 2^{m+l-1}$. Substituting m for $m+l$ gives

$$-2^{m-1} < \sum_{i \leq m-1} k_i \cdot 2^i < 2^{m-1} \tag{4}$$

We note that for the minimal m such that $k_m \neq 0$ we have $\sum_{i \leq m-1} k_i \cdot 2^i = 0$. Hence (4) holds for every m such that $k_m \neq 0$.

Because k_m is odd, we have $-(2^w - 1) \leq k_m \leq 2^w - 1$. Adding $k_m \cdot 2^m$ to (4) gives a slightly stronger version of (3):

$$-(2^{m+w} - 2^{m-1}) < \sum_{i \leq m} k_i \cdot 2^i < 2^{m+w} - 2^{m-1} \tag{5}$$

One consequence of subtracting negative wNAF components is that the wNAF representation may be one digit longer than the binary representation of the number. For n-digits binary numbers Möller [19] suggests using $k_i \leftarrow \lfloor k \rfloor_{2^w}$ when $i = n - w - 1$ and e is odd, where $\lfloor \cdot \rfloor_x$ denotes the reduction modulo x into the interval $[0, \ldots, x)$. This avoids extending the wNAF representation in half the cases at the cost of weakening the non-adjacency property of the representation.

2.3 Lattice background

Before we describe how to get the necessary information from the side-channel attack, we recall from previous works what kind of information we are looking for. As in previous works [2, 4, 5, 14, 21, 22], the side-channel information is used to construct a lattice basis and the secret key is then retrieved by solving a lattice problem on this lattice.

Generally, for a private key α and a group order q, in previous works the authors somehow derive triples (t_i, u_i, z_i) from the side-channel information such that

$$-q/2^{z_i+1} < v_i = |\alpha \cdot t_i - u_i|_q < q/2^{z_i+1}. \tag{6}$$

Note that for arbitrary α and t_i, the values of v_i are uniformly distributed over the interval $[-q/2, q/2)$. Hence, each such triple provides about z_i bits of information about α. The use of a different z_i per equation was introduced in [2]. If we take d such triples we can construct the following lattice basis

$$B = \begin{pmatrix} 2^{z_1+1} \cdot q & & & \\ & \ddots & & \\ & & 2^{z_d+1} \cdot q & \\ 2^{z_1+1} \cdot t_1 & \dots & 2^{z_d+1} \cdot t_d & 1 \end{pmatrix},$$

whose rows generate the lattice that we use to retrieve the secret key. Now consider the vector $\mathbf{u} = (2^{z_1+1} \cdot u_1, \dots, 2^{z_d+1} \cdot u_d, 0)$, which consists of known quantities. Equation (6) implies the existence of integers $(\lambda_1, \dots, \lambda_d)$ such that for the vectors $\mathbf{x} = (\lambda_1, \dots, \lambda_d, \alpha)$ and $\mathbf{y} = (2^{z_1+1} \cdot v_1, \dots, 2^{z_d+1} \cdot v_d, \alpha)$ we have

$$\mathbf{x} \cdot B - \mathbf{u} = \mathbf{y}.$$

Again using Equation (6), we see that the 2-norm of the vector $\mathbf{y}$ is at most $\sqrt{d \cdot q^2 + \alpha^2} \approx \sqrt{d+1} \cdot q$. Because the lattice determinant of $L(B)$ is $2^{d+\sum z_i} \cdot q^d$, the lattice vector $\mathbf{x} \cdot B$ is heuristically the closest lattice vector to $\mathbf{u}$. By solving the Closest Vector Problem (CVP) on input of the basis B and the target vector $\mathbf{u}$, we obtain $\mathbf{x}$ and hence the secret key α.

There are two important methods of solving the closest vector problem: using an exact CVP-solver or using the heuristic embedding technique to convert it to a Shortest Vector Problem (SVP). Exact CVP-solvers require exponential time in the lattice rank ($d+1$ in our case), whereas the SVP instance that follows from the embedding technique can sometimes be solved using approximation methods that run in polynomial time. Because the ranks of the lattices in this work become quite high when attacking a 521 bit key, we mostly focus on using the embedding technique and solving the associated SVP instance in this case.

The embedding technique transforms the previously described basis B and target vector $\mathbf{u}$ to a new basis B', resulting in a new lattice of dimension one higher than that generated by B:

$$B' = \begin{pmatrix} B & 0 \\ \mathbf{u} & q \end{pmatrix},$$

Following the same reasoning as above, we can set $\mathbf{x}' = (\mathbf{x}, -1)$ and obtain the lattice vector $\mathbf{y}' = \mathbf{x}' \cdot B' = (\mathbf{y}, -q)$. The 2-norm of $\mathbf{y}'$ is upper bounded by approximately $\sqrt{d+2} \cdot q$, whereas this lattice has determinant $2^{d+\sum z_i} \cdot q^{(d+1)}$. Note, however, that this lattice also contains the vector

$$(-t_1, \dots, -t_d, q, 0) \cdot B' = (0, \dots, 0, q, 0)$$

which will most likely be the shortest vector of the lattice. Still, our approximation algorithms for SVP work on bases and it is obvious to see that any basis of the same lattice must contain a vector ending in $\pm q$. Thus, it is heuristically likely that the resulting basis contains the short vector $\mathbf{y}'$, which reveals α.

To summarize, we turn the side-channel information into a lattice and claim that, heuristically, finding the secret key is equivalent to solving a CVP instance. Then, we claim that, again heuristically, solving this CVP instance is equivalent to solving an SVP instance using the embedding technique. In Section 5 we will apply the attack to simulated data to see whether these heuristics hold up.

3 Using the wNAF Information

Assuming we have a side channel that leaks the double and add chain of the scalar multiplication. We know how to use the leaked LSBs [2]. These leaked LSBs carry, on average, two bits of information.

Given a double and add chain, the positions of the add operations in the chain correspond to the non-zero digits in the wNAF representation of the ephemeral key k. Roughly speaking, in half the cases the distance between consecutive non-zero digits is $w+1$. In a quarter of the cases it is $w+2$ and so on. Hence, the average distance between consecutive non-zero digits is $w+\sum_i i/2^i = w+2$. Since there are 2^w non-zero digits, we expect that the double and add chain carries two bits of information per each non-zero digit position.

Reducing this information to an instance of the HNP presents three challenges:

- The information is not consecutive, but is spread along the scalar.
- Due to the use of negative digits in the wNAF representation, the double and add chain does not provide direct information on the bits of the scalar
- Current techniques lose half the information when the information is not at the beginning or end of the scalar.

As described in [2], the OpenSSL implementation departs slightly from the descriptions of ECDSA in Section 1. As a countermeasure to the Brumley and Tuveri remote timing attack [5], OpenSSL adds q or $2 \cdot q$ to the randomly chosen ephemeral key, ensuring that k is $n+1$ bits long. While the attack is only applicable to curves defined over binary fields, the countermeasure is applied to all curves. Consequently, our analysis assumes that $2^n \leq k < 2^{n+1}$.

To handle non-consecutive information, we extract a separate HNP instance for each consecutive set of bits, and use these in the lattice. The effect this has on the lattice attack is discussed in Section 4.

To handle the indirect information caused by the negative digits in the wNAF representation we find a linear combination of k in which we know the values of some consecutive bits, we can use that to build an HNP instance.

Let m and $m+l$ be the positions of two consecutive non-zero wNAF digits where $m+l < n$. From the definition of the wNAF representation we know that $k = \overline{k_{m+l}} 2^{m+l} + \sum_{i \leq m} k_i 2^i$. We can now define the following values:

$$a = \frac{\overline{k_{m+l}} - 1}{2}$$
$$c = \sum_{i \leq m} k_i \cdot 2^i + 2^{m+w}$$

By (5) we have

$$2^{m-1} < c < 2^{m+w+1} - 2^{m-1} \tag{7}$$

From (2) we have

$$k - 2^{m+l} + 2^{m+w} = a \cdot 2^{m+l+1} + c$$

where $0 \leq a < 2^{n-m-l}$ and because $l \geq w+1$ there are $l-w$ consecutive zero bits in $k - 2^{m+l} + 2^{m+w}$.

In order to extract this information, we rely on a property of the curve where the group order q is close to a power of two. More precisely, $q = 2^n - \varepsilon$ where $|\varepsilon| < 2^p$ for $p \approx n/2$. We note that many of the standard curves have this property.

Let $K = A \cdot 2^n + C$, with $0 \leq A < 2^{L_1}$ and $2^{p+L_1} \leq C < 2^{L_1+L_2} - 2^{p+L_1}$, note that this implies $L_2 > p$. Because $q = 2^n - \varepsilon$ we get $K - A \cdot q = K - A \cdot 2^n + A \cdot \varepsilon = C + A \cdot \varepsilon$. Now, $|\varepsilon| < 2^p$. Consequently, $0 \leq K - A \cdot q < 2^{L_1+L_2}$ and we get $\left|K - 2^{L_1+L_2-1}\right|_q < 2^{L_1+L_2-1}$. For $p+1 < m < n-l$ we can set

$$L_1 = n - m - l$$
$$L_2 = m + w$$
$$C = c \cdot 2^{n-m-l-1} = c \cdot 2^{L_1-1}$$
$$K = (k - 2^{m+l} + 2^{m+w}) \cdot 2^{n-m-l-1} = (k - 2^{m+l} + 2^{m+w}) \cdot 2^{L_1-1} = a \cdot 2^n + C$$

From (7) we obtain $2^{L_1+m-2} < C < 2^{L_1+L_2} - 2^{L_1+m-2}$ which, because $m \geq p-2$, becomes $2^{p+L_1} < C < 2^{L_1+L_2} - 2^{p+L_1}$. Thus, we have

$$\left|(k - 2^{m+l} + 2^{m+w}) \cdot 2^{n-m-l-1} - 2^{n-l+w-1}\right|_q < 2^{n-l+w-1}$$

Noting that $k = \alpha \cdot r \cdot s^{-1} + h \cdot s^{-1} \pmod{q}$, we can define the values

$$\begin{aligned} t &= \lfloor r \cdot s^{-1} \cdot 2^{n-m-l-1} \rfloor_q, \\ u &= \lfloor 2^{n+w-l-1} - (h \cdot s^{-1} + 2^{m+w} - 2^{m+l}) \cdot 2^{n-m-l-1} \rfloor_q, \\ v &= |\alpha \cdot t - u|_q. \end{aligned}$$

$|v| \leq 2^{n-l+w-1} \approx q/2^{l-w+1}$, which gives us an instance of the HNP which carries $l-w$ bits of information.

4 Heuristic Analysis

Now we know how to derive our triples t_i, u_i and z_i that are used to construct the lattice. The next obvious question is: How many do we need before we can retrieve the private key α? Because the lattice attack relies on several heuristics, it is hard to give a definitive analysis. However, we will give heuristic reasons here, similar to those for past results.

Each triple (t_i, u_i, z_i) gives us z_i bits of information. If this triple comes from a pair (m, l) such that $p+1 < m < n-l$, then $z_i = l - w$. In Section 3 we know that on average $l = w+2$. Since the positions of the non-zero digits are independent of p, on average we lose half the distance between non-zero digits, or $(w+2)/2$ bits, before the first usable triple and after the last usable triple, which leaves us with $n-1-(p+2)-(w+2)$ bits where our triples can be. The average number of triples is now given by $(n-p-3-(w+2))/(w+2)$ and each of these triples gives us $l-w=2$ bits on average. Combining this yields $2 \cdot (n-p-3-(w+2))/(w+2) = 2 \cdot (n-p-3)/(w+2) - 2$ bits per signature. For the **secp256k1** curve we have that $n = 256$, $p = 129$ and $w = 3$, leading to 47.6 bits per signature on average. Our data obtained from perfect side-channels associated to 1001 signatures gives us an average of 47.6 with a 95% confidence interval of ± 0.2664. For the **secp521r1** curve, we have that $n = 521$, $p = 259$ and $w = 4$, which suggests 84.33 bits per signature on average. The data average here is 84.1658 with a 95% confidence interval of ± 0.3825. See also the $Z = 1$ cases of Figures 1 and 2, which show the distribution of the bits leaked per signature in the 256-bit and 521-bit cases, respectively.

This formula suggests that on average, six signatures would be enough to break a 256-bit key (assuming a perfect side channel), since $47.6 \cdot 6 = 285.6 > 256$. However, in our preliminary experiments the attack did not succeed once when using six or even seven signatures. Even eight or nine signatures gave a minimal success probability. This indicates that something is wrong with the heuristic. In general there are two possible reasons for failure. Either the lattice problem has the correct solution but it was too hard to solve, or the solution to the lattice problem does not correspond to the private key α. We will now examine these two possibilities and how to deal with them.

4.1 Hardness of the lattice problem

Generally, the lattice problem becomes easier when adding more information to the lattice, but it also becomes harder as the rank increases. Since each triple adds information but also increases the rank of the lattice, it is not always clear whether adding more triples will solve the problem or make it worse. Each triple contributes z_i bits of information, so we would always prefer triples with a higher z_i value. Therefore, we set a bound $Z \geq 1$ and only keep those triples that have $z_i \geq Z$. However, this decreases the total number of bits of information we obtain per signature. If Z is small enough, then roughly speaking we only keep a fraction 2^{1-Z} of the triples, but now each triple contributes $Z+1$ bits on average. Hence, the new formula of bits per signature becomes

$$2^{1-Z} \cdot (Z+1) \cdot ((n-p-3)/(w+2) - 1).$$

Our data reflects this formula as well as can be seen in Figures 1 and 2 for the 256-bit and the 521-bit cases, respectively. In our experiments we will set an additional bound d on the number of triples we use in total, which limits the lattice rank to $d+1$. To this end, we sort the triples by z_i and then pick the first d triples to construct the lattice. We adopt this approach for our experiments and the results can be found in Section 5.

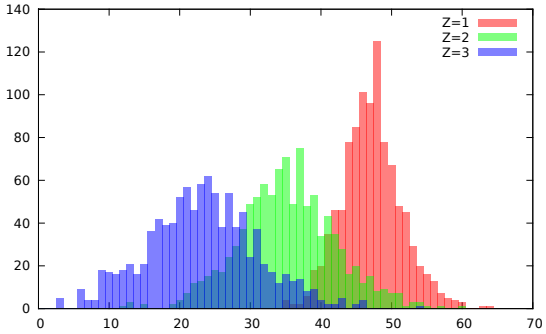

Fig. 1: Number of signatures against bits per signature in the 256 bit case.

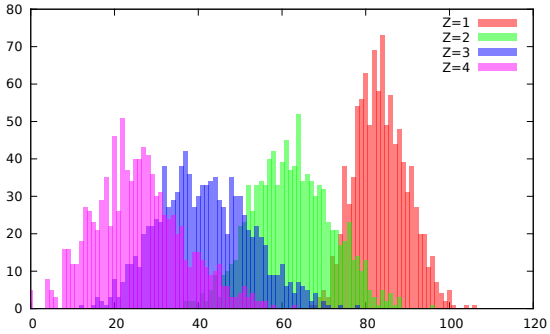

Fig. 2: Number of signatures against bits per signature in the 521 bit case.

4.2 Incorrect solutions

The analysis of Nguyen and Shparlinski [22] requires that the t_i values in the triples are taken uniformly and independently from a distribution that satisfies some conditions. However, it is easy to see that when two triples are taken from the same signature, the values for the $t_i = \lfloor r \cdot s^{-1} \cdot 2^{n-m_i-l_i-1} \rfloor_q$ and $t_j = \lfloor r \cdot s^{-1} \cdot 2^{n-m_j-l_j-1} \rfloor_q$ are not even independent, as they differ $\bmod q$ by a factor that is a power of 2 less than 2^n.

Recall from Sections 2.3 and 3 how the triples are used and created, respectively. Consider a triple (t_{ij}, u_{ij}, z_{ij}) corresponding to a signature (r_i, s_i, h_i). The corresponding $v_{ij} = |\alpha \cdot t_{ij} - u_{ij}|_q$ satisfies

$$\begin{aligned} |v_{ij}| &= \Big| |\alpha \cdot (r_i \cdot s_i^{-1} \cdot 2^{n-m_j-l_j-1}) - 2^{n+w-l_j-1} \\ &\qquad + (h_i \cdot s_i^{-1} + 2^{m_j+w} - 2^{m_j+l}) \cdot 2^{n-m_j-l_j-1}|_q \Big| \\ &\leq q/2^{z_{ij}+1}, \end{aligned}$$

which is equivalent to

$$|v_{ij}| = \Big| |(\alpha \cdot r_i + h_i) \cdot s_i^{-1} \cdot 2^{n-m_j-l_j-1} - 2^{n-1}|_q \Big| \leq q/2^{z_{ij}+1},$$

where $p+1 < m_j < n-l_j$ and $z_{ij} = l-w$. Now $(\alpha \cdot r_i + h_i) \cdot s_i^{-1} = k_i \mod q$ and we know that the previous statement holds due to the structure of k_i, specifically due to its bits $m_j+w, \ldots, m_j+l_j-1$ repeating, with bit m_j+l_j being different than the preceding bit. But the map $x \mapsto (x \cdot r_i + h_i) \cdot s_i^{-1}$ is a bijection $\bmod q$, and hence for each i there will be many numbers X such that for all j

$$|v_{ij}(X)| = \Big| |(X \cdot r_i + h_i) \cdot s_i^{-1} \cdot 2^{n-m_j-l_j-1} - 2^{n-1}|_q \Big| \leq q/2^{z_{ij}+1}.$$

Let $S_i = \{X : |v_{ij}(X)| \leq q/2^{z_{ij}+1} \text{ for all } j\}$. If we now have that there exists an $X \in \bigcap_i S_i$ such that

$$X^2 + \sum_{i,j} (2^{z_{ij}} \cdot v_{ij}(X))^2 < \alpha^2 + \sum_{i,j} (2^{z_{ij}} \cdot v_{ij}(\alpha))^2,$$

then it is very unlikely that the lattice algorithm will find α, because X corresponds to a better solution to the lattice problem. Note that this problem arises when fewer signatures are used, because this leads to fewer distinct values for (r_i, s_i, h_i) and hence fewer sets S_i that need to intersect. This suggests that increasing the number of signatures could increase the success probability.

Assuming that the S_i are random, we want to determine what is the probability that their intersection is non-empty. First we consider the size of the S_i. Recall that S_i consists of all $X \bmod q$ such that $v_{ij}(X)$ has 'the same structure as

k_i'. This means that for each triple specified by m_j and l_j, the bits $m_j+w,\ldots,m_j+l_j-1$ repeat, and bit m_j+l_j is the opposite of the preceding bits. There are approximately $2^{n-(l_j-w+1)+1}$ numbers $\bmod q$ that have this structure. Let f_i be the number of triples of signature i and $g_{ij}=(l_j-w+1)$ be the number of bits fixed by triple j of signature i. Then, because the triples do not overlap and because $v_{ij}(.)$ is a bijection, we have that

$$\log_2(|S_i|)=n-\sum_{j=1}^{f_i}(1-g_{ij})=n-f_i+\sum_{j=1}^{f_i}g_{ij}.$$

Let $s_i=|S_i|$ and assume that the S_i are chosen randomly and independently from all the subsets of integers in the range $[0,\ldots,N-1]$ (of size s_i), where $N=2^n$. Consider the following probability

$$p_i=\mathbb{P}(0\in S_i)=s_i/N,$$

since S_i is randomly chosen. Now, because the S_i are also chosen independently, we have

$$\mathbb{P}\left(0\in\bigcap_i S_i\right)=\prod_i p_i.$$

Finally, since this argument holds for any $j\in[0,\ldots,N-1]$, we can apply the union bound to obtain

$$p_{\text{fail}}=\mathbb{P}\left(\bigcup_j\left(j\in\bigcap_i S_i\right)\right)\leq\sum_j\mathbb{P}\left(0\in\bigcap_i S_i\right)=N\cdot\prod_i p_i. \tag{8}$$

Recall that each signature has $f_i=2^{1-Z}\cdot((n-p-3)/(w+2)-1)$ triples on average and each triple contributes $Z+1$ bits on average, which means $g_{ij}=Z+2$ on average. If we plug in the numbers $n=256$, $p=129$, $w=3$ and $Z=3$, we get that $f_i\approx 6$, $g_{ij}=5$ and hence $p_i\approx 2^{-6\cdot(5-1)}\approx 2^{-24}$ if we assume an average number of triples and bits in each signature. This in turn gives us an upper bound of $p_{\text{fail}}\leq N/2^{24\cdot k}$. If $k\geq 11$, this upper bound is less than one, so this clearly suggests that from about eleven signatures and up, we should succeed with some probability, which is indeed the case from our experiments.

Repeating this for $n=521$, $p=259$, $w=4$ and $Z=4$, we obtain $f_i\approx 5$, $g_{ij}=6$ and hence $p_i\approx 2^{-5\cdot(6-1)}\approx 2^{-25}$. Consequently, $p_{\text{fail}}\leq N/2^{25\cdot k}$, which is less than one when $k\geq 21$. However, in our experiments we require at least 30 signatures to obtain the secret key with some probability. Thus the above analysis is only approximate as the secret key length increases.

5 Results With a Perfect Side-Channel

Subsection 2.3 outlined our (heuristic) approach to obtain the secret key from a number of triples (t_i,u_i,z_i) using lattices and Section 3 outlined how to generate these triples from the side-channel information. In this section we will look at some experimental results to see if our heuristic assumptions are justified.

As per Section 4, we used the following approach for our experiments. First, we fix a number of signatures s, a lattice rank d and a bound Z. We then take s signatures at random from our data set and derive all triples such that $z_i\geq Z$, sorting them such that the z_i are in descending order. If we have more than d triples, we only take the first d to construct the lattice. Finally we attempt to solve the lattice problem and note the result. All executions were performed in single thread on an Intel Core i7-3770S CPU running at 3.10 GHz.

When solving the CVP instances there are three possible outcomes. We obtain either no solution, the private key or a wrong solution. No solution means that the lattice problem was too hard for the algorithm and constraints we used, but spending more time and using stronger algorithms might still solve it. When a 'wrong' solution is obtained, this means that our heuristics failed: the solution vector was not unique, in the sense that there were other lattice vectors within the expected distance from our target vector.

When solving the SVP instance there are only two outcomes. Either we obtain the private key or not. However, in this case it is not as clear whether a wrong solution means that there were other solutions due to the additional heuristics involved. The complete details of our experimental data are given in the Appendix.

5.1 256 bit key

For the 256 bit case, we used BKZ with block size 20 from fplll [6] to solve the SVP instances, as well as to pre-process the CVP instances. To solve the CVP, we applied Schnorr-Euchner enumeration [23] using linear pruning [12] and limiting the number of enumerated nodes to 2^{29}.

The CVP approach seems the best, as the lattice rank $(d+1)$ remains quite small. We restrict our triples to $Z = 3$ to keep the rank small, but a smaller Z would not improve our results much. See the appendix for details. We observed that failures are mostly caused by 'wrong' solutions in this case, rather than the lattice problem being too hard. In all cases we found that using 75 triples gave the best results. Table 2 in the Appendix lists the runtimes and success probabilities of the lattice part of the attack for varying s. The results are graphically presented in Figures 4 and 5 in the Appendix.

5.2 521 bit key

For the 521 bit case, we used BKZ with block size 20 from fplll [6] to solve the SVP instances. Due to the higher lattice ranks in this case, solving the CVP instances proved much less efficient, even when restricting the triples to $Z = 4$.

With 30 signatures we get a small probability of success in the lattice attack whereas with 40 signatures we can obtain the secret key in more than half of the cases. It should be noted that as the number of signatures increases, the choice of d becomes less important, because the number of triples with more information increases. See the Appendix for Table 4 details and Figures 6 and 7 for a graphical representation.

6 Results in a Real-Life Attack

So far our discussion was based on the assumption of a perfect side-channel. That is, we assumed that the double-and-add chains are recovered without any errors. Perfect side-channels are, however, very rare. In this section we extend the results to the actual side-channel exposed by the FLUSH+RELOAD technique.

The attack was carried on an HP Elite 8300, running CentOS 6.5. The victim process runs OpenSSL 1.0.1f, compiled to include debugging symbols. These symbols are not used at run-time and do not affect the performance of OpenSSL. We use them because they assist us in finding the addresses to probe by avoiding reverse engineering [9]. The spy uses a time slot of 1,200 cycles (0.375μs). In each time slot it probes the memory lines containing the last field multiplication within the group add and double functions. (`ec_GFp_simple_add` and `ec_GFp_simple_dbl`, respectively.) Memory lines that contain function calls are accessed both before and after the call, reducing the chance of a spy missing the access due to overlap with the probe. Monitoring code close to the end of the function eliminates false positives due to speculative execution. See Yarom and Falkner [27] for a discussion of overlaps and speculative execution.

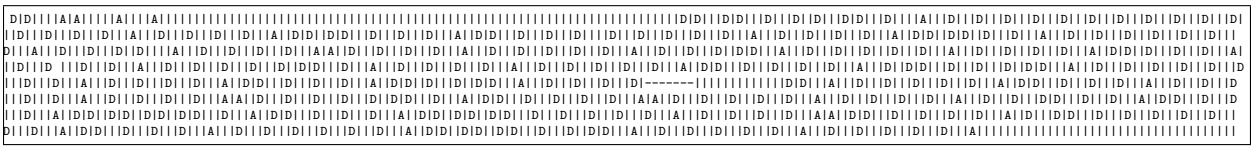

Fig. 3: FLUSH+RELOAD spy output. Vertical bars indicate time-slot boundaries; 'A' and 'D' are probes for OpenSSL access to add and double; dashes indicate missed time-slots.

Figure 3 shows an example of the output of the spy when OpenSSL signs using **secp256k1**. The double and three addition operations at the beginning of the captured sequence are the calculation of the pre-computed wNAF digits. Note the repeated capture of the double and add operations due to monitoring a memory line that contains a function call. The actual wNAF multiplication starts closer to the end of the line, with 7 double operations followed by a group addition.

In this example, the attack captures most of the double and add chain. It does, however, miss a few time-slots and consequently a few group operations in the chain. The spy recognises missed time-slots by noting inexplicable gaps in the processor cycle counter. As we do not know which operations are missed, we lose the bit positions of the operations that precede the missed time-slots. We believe that the missed time-slots are due to system activity which suspends the spy.

Occasionally OpenSSL suspends the calculation of the scalar multiplication to perform memory management functions. These suspends confuse our spy program, which assumes that the scalar multiplication terminated. This, in turn, results in a short capture, which cannot be used for the lattice attack.

To test prevalence of capture errors we captured 1,000 scalar multiplications and compared the capture results to the ground truth. 342 of these captures contained missed time-slots. Another 77 captures contains less than 250 group operations and are, therefore, too short. Of the remaining 581 captures, 577 are perfect while only four contain errors that we could not easily filter out.

Recall, from Section 5, that 13 perfectly captured signatures are sufficient for breaking the key of a 256 bits curve with over 50% probability. An attacker using FLUSH+RELOAD to capture 25 signatures can thus expect to be able to filter out 11 that contain obvious errors, leaving 14 that contain no obvious errors. With less than 1% probability that each of these 14 captures contains an error, the probability that more than one of these captures contains an error is also less than 1%. Hence, the attacker only needs to test all the combination of choosing 13 captures out of these 14 to achieve a 50% probability of breaking the signing key.

Several optimisations can be used to improve the figure of 25 signatures. Some missed slots can be recovered and the spy can be improved to correct short captures. Nevertheless, it should be noted that this figure is still an order of magnitude than the previously best known result of 200 signatures [2], where 200 signatures correspond to a 3.5% probability of breaking the signing key, whereas 300 signatures were required to get a success probability greater than 50%.

Acknowledgements

The authors would like to thank Ben Sach for helpful conversations during the course of this work. The first and second authors work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant EP/I03126X, and by Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079[3].

The third author wishes to thank Dr Katrina Falkner for her advice and support and the Defence Science and Technology Organisation (DSTO) Maritime Division, Australia, who partially funded his work.

References

1. American National Standards Institute. *ANSI X9.62, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm*, 1999.
2. Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh aah..., just a little bit": A small amount of side channel can go a long way. In Lejla Batina and Matthew Robshaw, editors, *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 8731 of *Lecture Notes in Computer Science*, pages 75–92, Busan, Korea, September 2014. Springer.
3. Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes. In *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 129–142, 1996.
4. Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 667–684. Springer-Verlag, 2009.
5. Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Diaz, editors, *Computer Security - ESORICS 2011*, pages 355–371, Leuven, Belgium, September 2011.

[3] The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

6. David Cadé, Xavier Pujol, and Damien Stehlé. FPLLL-4.0.4. `http://perso.ens-lyon.fr/damien.stehle/fplll/`, 2013.
7. Certicom Research. *SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2.0*, January 2010.
8. Mathieu Ciet and Marc Joye. (virtually) free randomization techniques for elliptic curve cryptography. In Sihan Qing, Dieter Gollmann, and Jianying Zhou, editors, *Proceedings of the fifth International Conference on Information and Communications Security*, volume 2836 of *Lecture Notes in Computer Science*, pages 348–359, Huhehaote, China, October 2003. Springer.
9. Teodoro Cipresso and Mark Stamp. Software reverse engineering. In Peter Stavroulakis and Mark Stamp, editors, *Handbook of Information and Communication Security*, chapter 31, pages 659–696. Springer, 2010.
10. Richard E. Crandall. Method and apparatus for public key exchange in a cryptographic system. US Patent 5,159,632, October 1992.
11. Benoit Feix, Mylène Roussellet, and Alexandre Venelli. Side-channel analysis on blinded regular scalar multiplications. In Willi Meier and Debdeep Mukhopadhyay, editors, *Proceedings of the 15th International Conference on Cryptology in India (INDOCRYPT 2014)*, Lecture Notes in Computer Science, New Delhi, India, December 2014. Springer.
12. Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 257–278, 2010.
13. Daniel M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, April 1998.
14. Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, 2001.
15. Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume one, pages 225–230, Ottawa, Ontario, Canada, June 2007.
16. Arjen K. Lenstra, Hendrik W. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515 –534, 1982.
17. Lixin Li, James E. Just, and R. Sekar. Address-space randomization for Windows systems. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 329–338, Miami Beach, Florida, United States, December 2006.
18. Bodo Möller. Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks. In Agnes Hui Chan and Virgil D. Gligor, editors, *Proceedings of the fifth International Conference on Information Security*, number 2433 in Lecture Notes in Computer Science, pages 402–413, São Paulo, Brazil, September 2002.
19. Bodo Möller. Improved techniques for fast exponentiation. In P. J. Lee and C. H. Lim, editors, *Information Security and Cryptology - ICISC 2002*, number 2587 in Lecture Notes in Computer Science, pages 298–312. Springer-Verlag, 2003.
20. National Institute of Standards and Technology. *FIPS PUB 186-4 Digital Signature Standard (DSS)*, 2013.
21. Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3):151–176, June 2002.
22. Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, Codes and Cryptography*, 30(2):201–217, September 2003.
23. Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In *Fundamentals of Computation Theory – FCT 1991*, volume 529 of *Lecture Notes in Computer Science*, pages 68–85. Springer, 1991.
24. Jerome A. Solinas. Generalized Mersenne numbers. Technical Report CORR-39, University of Waterloo, 1999.
25. Carl A. Waldspurger. Memory resource management in VMware ESX Server. In David E. Culler and Peter Druschel, editors, *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 181–194, Boston, Massachusetts, United States, December 2002.
26. Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, February 2014. `http://eprint.iacr.org/`.
27. Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, pages 719–732, San Diego, California, United States, August 2014.

A Experimental results

A.1 256 Bit Keys

		SVP		CVP	
s	d	Time (s)	p_{succ} (%)	Time (s)	p_{succ} (%)
10	60	1.47	0.0	1.56	0.5
10	65	1.42	1.0	1.90	2.5
10	70	1.44	1.5	2.45	4.0
10	75	1.50	1.5	2.25	7.0
11	60	1.28	0.0	1.63	0.5
11	65	1.68	5.0	2.35	6.5
11	70	1.86	2.5	3.15	19.0
11	75	2.05	7.5	4.66	25.0
11	80	2.12	6.0		
12	60	1.27	2.0	1.69	7.0
12	65	1.71	2.5	2.45	10.5
12	70	2.20	7.5	3.99	29.5
12	75	2.57	10.5	7.68	38.5
12	80	2.90	13.0		
12	85	3.12	8.5		
12	90	3.21	15.5		
13	60	1.30	3.5	1.92	8.5
13	65	1.77	6.0	2.79	25.5
13	70	2.39	11.0	4.48	46.5
13	75	3.16	19.0	11.30	54.0
13	80	3.67	18.5		
13	85	3.81	21.5		
13	90	4.37	25.0		

Table 1: Results for d triples taken from s signatures with a 256-bit key ($Z = 3$)

s	Time (s)	p_{succ} (%)
10	2.25	7.0
11	4.66	25.0
12	7.68	38.5
13	11.30	54.0

Table 2: CVP results for 75 triples taken from s signatures with a 256-bit key ($Z = 3$)

A.2 521 Bit Keys

s	d	Time (s)	p_{succ} (%)
30	130	50.10	4.0
30	135	58.80	3.0
30	140	66.65	3.5
30	145	69.68	2.5
32	130	50.15	6.5
32	135	58.07	6.5
32	140	62.55	4.0
32	145	67.46	5.0
32	150	70.77	9.5
34	130	50.00	15.5
34	135	55.93	10.5
34	140	62.83	16.0
34	145	64.41	14.0
34	150	70.50	16.0
34	155	71.07	11.5
36	130	48.71	24.5
36	135	54.74	21.0
36	140	59.25	22.5
36	145	62.32	29.0
36	150	65.60	29.0
36	155	68.57	24.5
38	130	49.04	38.5
38	135	53.86	36.0
38	140	57.14	38.5
38	145	61.31	42.5
38	150	66.75	36.5
38	155	66.52	36.5
40	130	47.73	53.0
40	135	50.80	49.0
40	140	54.88	52.0
40	145	60.47	47.0
40	150	64.77	53.0
40	155	64.95	52.5

s	d	Time (s)	p_{succ} (%)
31	130	48.50	7.5
31	135	59.91	3.5
31	140	67.35	6.0
31	145	69.96	5.5
33	130	49.70	8.0
33	135	56.52	11.5
33	140	60.31	11.5
33	145	66.39	8.5
33	150	70.54	13.5
33	155	75.49	8.5
35	130	49.76	12.0
35	135	55.33	24.5
35	140	59.50	15.5
35	145	65.59	19.5
35	150	66.93	24.0
35	155	69.67	20.0
37	130	48.20	24.0
37	135	54.79	23.5
37	140	58.60	28.0
37	145	60.05	29.0
37	150	63.40	27.5
37	155	69.14	34.5
39	135	50.99	45.5
39	140	58.81	46.0
39	145	57.08	47.5
39	150	62.35	41.5
39	155	64.99	42.5

Table 3: SVP results for d triples taken from s signatures with a 521-bit key ($Z = 4$)

s	d	Time (s)	p_{succ} (%)
30	130	50.10	4.0
31	130	48.50	7.5
32	150	70.77	9.5
33	150	70.54	13.5
34	140	62.83	16.0
35	135	55.33	24.5
36	145	62.32	29.0
37	155	69.14	34.5
38	145	61.31	42.5
39	145	57.08	47.5
40	130	47.73	53.0

Table 4: SVP results for d triples taken from s signatures with a 521-bit key ($Z = 4$)

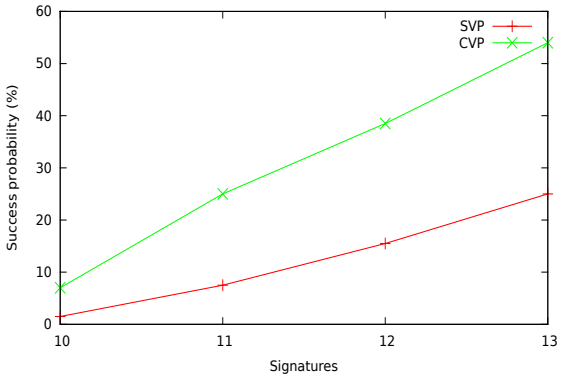

Fig. 4: Success probability per number of signatures against a 256 bit key

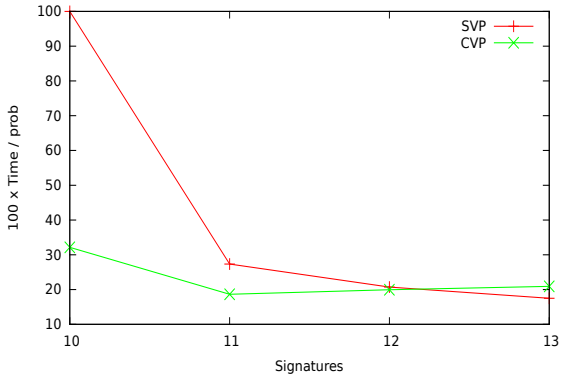

Fig. 5: Expected running time per number of signatures against a 256 bit key

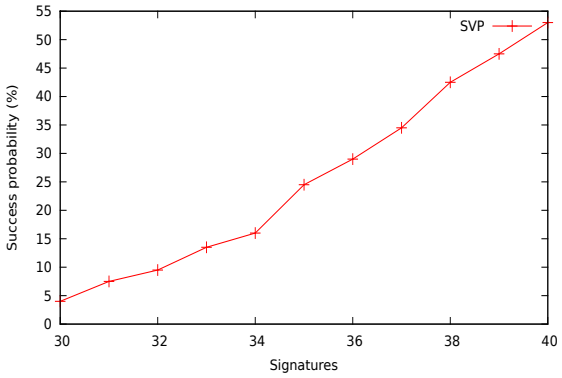

Fig. 6: Success probability per number of signatures against a 521 bit key

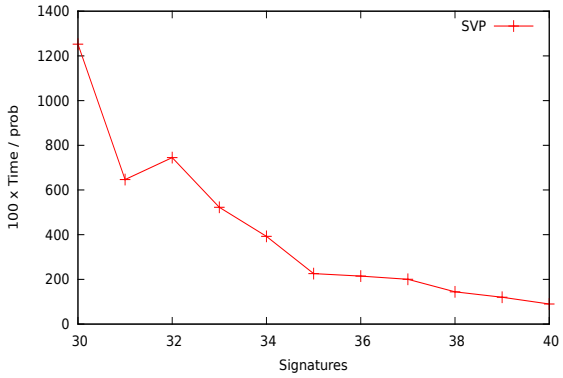

Fig. 7: Expected running time per number of signatures against a 521 bit key